The Dissertation Committee for Nitya Ranganathan

certifies that this is the approved version of the following dissertation:

Control Flow Speculation for Distributed Architectures

Committee:

Douglas C. Burger, Supervisor

Stephen W. Keckler

Kathryn S. Mckinley

Yale N. Patt

Daniel A. Jiménez

Control Flow Speculation for Distributed Architectures

by

Nitya Ranganathan, B.E.; M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2009

To my parents.

Acknowledgments

First and foremost, I would like to thank my advisor Doug Burger for giving me his guidance, support, and advice on numerous occasions. When I started as a graduate student several years ago, he taught me how to approach architecture research. He was always enthusiastic about positive results and encouraging whenever there was a disappointment. I learned many important things from him: looking for the upper-bounds when getting started, thinking about a two-line summary of why a paper would be cited in future, considering alternative solutions and so on. Doug gave me interesting problems to work on and provided me with ideas to explore alternative solutions. I will always be grateful to him for his encouragement during the numerous ups and downs over the years.

I thank Steve Keckler for his valuable guidance and support. I had the opportunity to work with him during the TRIPS prototype work and learned several things from design simplification for easier verification to writing Verilog code. From his feedback for paper drafts and during discussions, I learned to take a step back and think about the big picture. I thank my other dissertation committee members Kathryn McKinley, Yale Patt, and Daniel Jiménez for their feedback and comments on my thesis as well as initiating discussions during my proposal and defense talks which led to better writing and experiments.

In the first few years, I worked with Ramadass Nagarajan, Daniel Jiménez, and Calvin Lin along with Doug and Steve on hyperblock prediction. It was a good learning experience for my later years. During the TRIPS prototype chip development, I had the great opportunity to interact with Chuck Moore and Robert McDonald which helped me

during the initial predictor design phase. The regular discussions with the entire prototype team taught me lessons that will be useful for any large projects in future. I also thank Bill Yoder for his efforts at helping me use the toolchain more easily.

The students in the CART research group were always interested in discussions, brainstorming, helping out with infrastructure issues, and simply lending a ear to help solve problems. It has a wonderful time being a part of this group. During the prototype chip work, I worked closely with Premkishore Shivakumar, Divya Gulati, and Ramadass Nagarajan. I have learned a lot from my interactions with them and working together as a group was always pleasant. For the TFlex work, I was part of a team including Changkyu Kim, Simha Sethumadhavan, and Sibi Govindan. From early brainstorming days to simulator development and paper writing, I was glad to be working with such a great team. I thank Sibi for giving me results from the prototype chip. I also wish to thank Bert Maher for tagging basic blocks for the correlation study and patiently letting me know the "best" compilation options every time. Several others have helped me in various ways for discussion of research ideas and simulator/infrastructure help including Karthikeyan Sankaralingam, Haiming Liu, Heather Hanson, Sadia Sharif, Suriya Subramaniam, Mark Gebhart, Katie Coons, and Aaron Smith. The interactions with all the CART students, especially the TRIPS prototype team members, have been very useful and interesting.

I thank the CS department staff members Gem Naivar, Gloria Ramirez, Katherine Utz, and Lydia Griffith for answering my numerous questions patiently, helping me handle forms and meet university deadlines, and in general, making graduate school life seem much easier.

I would like to thank Norm Jouppi for being my mentor during my internship at HP Labs. I learned several new things during the internship. I take this opportunity to thank the teachers and professors from my school and undergraduate days who were inspirations for my academic interests. I would like to especially thank my undergraduate thesis advisor Dr. Ranjani Parthasarathi who inspired me, taught me the fundamentals of computer

architecture, and gave me wonderful research guidance.

My friends from graduate school, especially Suju, Srujana, and Sibi provided food and gave me wonderful company on several evenings. I thank Karu for hosting me and guiding me for the first few days after my arrival in the US. I am also very thankful to my dear friends Arundhati, Meera, and Priyaa who constantly encouraged me despite being several time zones away.

I am extremely grateful to the constant support, advice, and kindness shown by my in-laws. They have always been very encouraging and I cannot thank them enough. During the graduate school life, my sister Swetha has provided all the love and encouragement that I can ever ask for. She has given me company on numerous nights from half way around the world, helped me make decisions, entertained me, and made me feel happy. I would like to thank her for always being there for me. I am very grateful to my best friend and husband Ram for everything he has done for me. We have known each other for more than a decade, and his unconditional love, support, and encouragement have only increased over the years. He has helped me in every way possible, more than I can ever list, and it is impossible to thank him adequately for everything. He has always been open to technical discussions, providing advice, and giving suggestions. He has had a busy time at graduate school and work, yet he has unfailingly stood by me through thick and thin. Finally, I thank my loving parents and grandmothers for everything they have done for me right from my childhood. It is impossible to even begin listing their sacrifices, constant encouragement, guidance, and help. My father's frequent inspirational talks, enquiries about my research, and advice on health were invaluable while my mother's support, encouragement, and belief in me made me bolder and stronger. They have made so many sacrifices for me and I am here because of them. I hope to give them some happiness by completing this dissertation.

NITYA RANGANATHAN

The University of Texas at Austin

May 2009

Control Flow Speculation for Distributed Architectures

Publication No. _____

Nitya Ranganathan, Ph.D.
The University of Texas at Austin, 2009

Supervisor: Douglas C. Burger

As transistor counts, power dissipation, and wire delays increase, the microprocessor industry is transitioning from chips containing large monolithic processors to multi-core architectures. The granularity of cores determines the mechanisms for branch prediction, instruction fetch and map, data supply, instruction execution, and completion. Accurate control flow prediction is essential for high performance processors with large instruction windows and high-bandwidth execution. This dissertation considers cores with very large granularity, such as TRIPS, as well as cores with extremely small granularity, such as TFlex, and explores control flow speculation issues in such processors. Both TRIPS and TFlex are

distributed block-based architectures and require control speculation mechanisms that can work in a distributed environment while supporting efficient block-level prediction, misprediction detection, and recovery.

This dissertation aims at providing efficient control flow prediction techniques for distributed block-based processors. First, we discuss simple exit predictors inspired by branch predictors and describe the design of the TRIPS prototype block predictor. Area and timing trade-offs in the predictor implementation are presented. We report the predictor misprediction rates from the prototype chip for the SPEC benchmark suite. Next, we look at the performance bottlenecks in the prototype predictor and present a detailed analysis of exit and target predictors using basic prediction components inspired from branch predictors. This study helps in understanding what types of predictors are effective for exit and target prediction. Using the results of our prediction analysis, we propose novel hardware techniques to improve the accuracy of block prediction. To understand whether exit prediction is inherently more difficult than branch prediction, we measure the correlation among branches in basic blocks and hyperblocks and examine the loss in correlation due to hyperblock construction. Finally, we propose block predictors for TFlex, a fully distributed architecture that uses composable lightweight processors. We describe various possible designs for distributed block predictors and a classification scheme for such predictors. We present results for predictors from each of the design points for distributed prediction.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As power dissipation and wire delays become first-order constraints, the microprocessor industry is moving from huge monolithic cores to chips containing a large number of smaller and simpler cores. Transistor counts have been steadily increasing for the last few decades in accordance with Moore's law [43]. Clock frequency, on the other hand has been scaling very slowly for the last few years. Future processors are expected to extract high performance from concurrent execution rather than clock frequency improvements. Hence there is a shift from poorly-scaling huge monolithic processors to highly partitioned architectures with an increasing number of cores on a chip. Partitioned structures have lower complexity and lesser power dissipation compared to monolithic structures. As wire delays and power increase and clock frequencies scale slowly, the extra transistors on a chip will be used to pack more cores. Software will play a key role in extracting parallelism from programs, utilizing multiple cores, and managing the mapping of cores to processes or threads. The challenges in the software design depend on the interface presented by the multi-core architecture and ability of the hardware to handle and manage the mapped threads.

In a multi-core processor, the core granularity and mix of cores play an important role in the design of the microarchitecture and protocols for instruction fetch and map, data supply, instruction execution, and completion. For example, a multi-core processor target-

ing the server-market may have 10's or 100's of lightweight cores each capable of handling one or two server threads. Complex mechanisms for speculation and out-of-order execution are not required in such processors. On the other hand, a versatile desktop processor may have a combination of some lightweight low-performance cores and some larger aggressive cores to handle powerful media and desktop applications. Processors targeting the high-performance market should be capable of executing several instructions every clock cycle and require high bandwidth fetch, execution, and completion. Typically, the enablers of high bandwidth fetch are control flow speculation and multiple instruction fetch using multiple instruction cache banks or block-level wide fetch. Similarly, the enablers of high bandwidth execution are renaming, dependence speculation, out-of-order execution, and large instruction windows.

Control speculation using branch prediction has been a significant contributor of high performance in modern microprocessors. A branch predictor guesses the direction of the branch and if predicted taken, computes or guesses the target of the branch. This prediction enables the fetch pipeline to redirect instruction fetch from the target of the branch without having to wait for the branch to resolve. There have been many branch prediction mechanisms proposed in the last few decades [25, 60, 69, 81]. The ultimate goal of a branch predictor is to provide highly accurate predictions in a simple design that has reasonable power, area, and latency. Modern processors often have simple single-cycle predictors (for example, a line predictor [3] or a bimodal predictor [69]) to quickly deliver predictions, backed up by a slower but more accurate predictor (for example, a PAs/Gshare hybrid predictor [41]).

TRIPS [56, 57] and TFlex [30, 31] are distributed block-based architectures having a highly partitioned scalable design without any large monolithic structures. Both architectures are multi-core processors with a tile-based design. While TRIPS has cores with large granularity, TFlex has cores with very small granularity called Composable Lightweight Processors (CLPs). Both follow a block-based and block-atomic execution model [42, 44],

to provide high fetch/execute/commit bandwidth and also amortize the overheads of speculation. TRIPS and TFlex use an Explicit Dataflow Graph Execution (EDGE) [2] Instruction Set Architecture (ISA). The compiler constructs large single-entry, multiple-exit blocks of instructions similar to hyperblocks (supporting predication) [38, 68]. The processor fetches and maps successive blocks on to the execution substrate using control speculation and executes the instructions within each block in a data flow fashion.

This dissertation explores control flow speculation mechanisms for distributed block-based architectures like TRIPS and TFlex. Both TRIPS and TFlex require control speculation mechanisms that can work in a distributed environment while supporting efficient block-level prediction, misprediction detection, and recovery. We look at designs of previously proposed branch and block predictors, analyze block predictability in hardware and the ability of various types of predictors to predict blocks, examine the loss of predictability in blocks, and propose monolithic block predictors for TRIPS and distributed block predictors for TFlex. The next few sections in this chapter give the overall context of our work and describe the thesis contributions.

1.1 Block-based, block-atomic and distributed Architectures

Conventional architectures such as superscalar and VLIW perform fetch, execute, and commit operations at the instruction granularity. Block-based architectures [14, 16, 42, 48, 78] perform similar operations at the block granularity. Hence, in a block-based processor, the unit of fetch, dispatch, execution, and commit is a block. In a block-atomic processor, atomicity of blocks is preserved: either the entire block is executed or it is entirely squashed. Misspeculation recovery and exception handling are done at block boundaries. In distributed architectures, various microarchitectural structures are partitioned and distributed across the core. For example, instructions can be fetched by a set of replicated fetch units while instructions are executed in a set of replicated execution units physically away from the fetch units.

The TRIPS architecture [56,57] is a distributed, tile-based organization that evolved in response to increasing wire delays seen in modern process technologies. The distributed architecture effectively tolerates wire delays through a combination of block-based, block-atomic execution [16, 44], and compiler-directed instruction placement and operand communication. TRIPS is the first architecture that uses an EDGE (Explicit Dataflow Graph Execution) ISA. EDGE architectures are defined by two main characteristics: direct operand communication among instructions in a block (dataflow model) and block-atomic execution. TFlex is an extension of the TRIPS architecture that makes it much more flexible to execute various types of workloads with high power efficiency and low area overhead.

The unit of execution in TRIPS and TFlex is the TRIPS hyperblock also referred to as a block. A hyperblock is a large single-entry multiple-exit block of possibly predicated instructions [39]. Several heuristics aid the block-construction process [38, 68]. Dynamically, the processor fetches and maps successive blocks on to the execution substrate using control-flow speculation. The instructions within each block are executed by the substrate in a data-flow fashion. Hence the architecture uses static placement (using the compiler scheduler) and dynamic issue of instructions depending on the availability of operands. Block-based execution in TRIPS provides high fetch, execute, and commit bandwidth. It also amortizes the overheads of speculation like renaming, register reads, and checkpointing microarchitectural state.

In TRIPS and TFlex, blocks of instructions fetched from the instruction cache banks are dispatched directly to the execution units where they are decoded and executed. A single Global Control Unit (GCU), also called Global Tile (GT) in TRIPS, manages the entire control for a block, namely, fetch, execute, flush, complete, and retire. Both TRIPS and TFlex use tile based designs with distributed protocols for instruction fetch, map, data supply, completion status, and commit. There is no single point of control that examines all instructions before making control decisions.

1.2 Control flow speculation and its importance

Control flow speculation uses branch prediction to predict the targets of branch instructions. These predicted targets are used by the fetch unit to steer instruction fetch. A high performance execution core requires a high performance fetch engine to feed instructions. Accurate control flow prediction is essential for keeping the fetch unit busy and reducing wasted cycles. Traditionally, branch prediction has been one of the key features that enable high performance. Without branch prediction, the front end has to either resort to waiting until the branch resolves or perform dual path or predicated execution. These techniques become inefficient when they are used for all branches in a program. Hence at least for a subset of branches, typically, a powerful branch predictor is used. There have been many branch prediction proposals in the last few decades and state-of-the-art branch predictors have reasonably high prediction accuracies [25, 60, 69, 81].

A branch predictor predicts the target of a branch. In other words, it determines the next basic block to be fetched once the current basic block is fetched. A block predictor, in general, predicts the address of the next block (could be any block of instructions such as a trace, hyperblock or a basic block) given the address of the current block. A control flow predictor, whether it is a branch predictor or a block predictor, is essential for high performance machines with high fetch bandwidth and large instruction windows.

Accurate control-flow speculation is equally (or more) important in TRIPS and TFlex (and other block-based high-performance architectures) as in general-purpose processors, since the cost of misspeculation is very high. Each block can potentially have up to 128 useful instructions. A control-flow misspeculation can cause several such blocks to be flushed from the pipeline, resulting in a heavy refill penalty and a lot of wasted effort. For effective control speculation in block-based processors, block-level prediction, misprediction detection, and recovery are necessary.

The next-block predictor is at the heart of the control flow speculation logic in the TRIPS/TFlex architecture. Any branch that fires in a TRIPS hyperblock leads to transfer

of control to another block and is referred to as an exit branch, or simply, an exit. A hyperblock can have several exits but exactly one of the exits fires. The next-block predictor guesses the target of the firing exit i.e., the address of the next block to be fetched in program order. While making a prediction, the block predictor does not get an opportunity to "see" the branches among the instructions fetched and dispatched to the execution units. Since the block predictor is physically away from the execution units, snooping the control network for branch instruction opcodes and other fields is not feasible. Hence, when a block is fetched, the predictor generally has only one piece of information about the block, namely the block address. Control information within the block, such as the number of branches, individual branch addresses, branch types, and targets are not available at the time of prediction. Making accurate predictions to fill the pipeline is more challenging than in conventional predictor designs.

Though control flow prediction is extremely important for high performance, it is not the only factor in providing efficient control speculation in a processor. Misprediction detection involves detecting a misprediction upon resolution of a branch instruction and the time it takes to detect a potential misprediction is crucial for performance. Similarly misprediction recovery, where the pipeline is flushed and a safe checkpointed state is restored for all relevant structures has to be efficient. If there are bottlenecks in misprediction detection or recovery, there is a significant loss in performance. This dissertation considers only control flow prediction and does not concentrate on exploring or enhancing misprediction detection and recovery.

1.3 Thesis statement

This dissertation aims at identifying and solving the challenges involved in doing distributed block prediction. We solve the new challenges imposed by a distributed block-atomic architecture and design and implement a block predictor that works reasonably well in the TRIPS prototype processor. We perform a systematic analysis of various basic predictors to

understand what types and combinations of predictors can capture the inherent predictability in blocks. Using these results, we propose hardware mechanisms for improving block prediction accuracy. Specifically, we propose methods to map state-of-the-art branch predictors to block prediction. We describe a novel exit predictor design that leverages the high accuracy of binary predictors. We also propose novel prediction components inspired by exit branch prediction. We make the observation that the inherent predictability that can be leveraged by these hardware techniques is still limited and propose a correlation analysis methodology that tracks the loss in predictability in hyperblocks compared to basic blocks. For the TFlex architecture, we introduce distributed prediction and present a classification scheme for distributed predictors. We propose predictors for each of the design points in the distributed prediction space.

1.4 Dissertation contributions

TRIPS and TFlex architecture research have been joint efforts of several people over the past decade. In this section, I highlight my specific contributions in the context of this dissertation.

- Though we were aware of the previous block prediction work, I experimented with several types of predictors to understand how exit branches could be predicted. This work culminated when we found a reasonably accurate hybrid exit predictor and compared its performance with similar predictors and other branch and block predictors. The best local/global exit predictor achieves a 5.3% misprediction rate compared to the 4.3% misprediction rate of a PAs/Gshare tournament hybrid branch predictor. Further, when compared to a branch predictor, the exit predictor reduces the total number of predictions by 72% while achieving 3.9% misses per thousand instructions (MPKI) for a 32 KB predictor.

- Using the exit predictor we finalized above, I developed the design for the TRIPS

prototype next block predictor comprising of a tournament exit predictor and a multi-component target predictor. I designed the predictor to match the new TRIPS architecture and microarchitecture specifications and block-atomicity support. The predictor has been designed as a three-cycle blocking predictor with support for speculative updates. Specifically, support for predicting branch types and dynamically learning return targets for return address prediction were unique design challenges. The predictor was implemented with area and timing optimizations. Measurements from the prototype chip indicate that the TRIPS block predictor achieves a mean misprediction rate of 11.5% for integer benchmarks and 4.3% for floating-point benchmarks.

- After the prototype predictor implementation, the new software toolchain development has progressed significantly. Using new versions of benchmark binaries, I did a retrospective analysis of the predictor to see whether there were bottlenecks in the design that led to low prediction rates compared to traditional branch predictors. We found that there were some significant bottlenecks in the exit and target predictor design. The exit predictor needed longer history lengths and larger prediction tables to make significantly better predictions. The offset widths in the target predictor components were insufficient to predict all branches accurately. The branch type predictor required more entries to improve type prediction accuracy. Finally, the presence of an indirect branch predictor could have significantly reduced the MPKI for some benchmarks.

- To improve block prediction, instead of taking the best predictors from past branch prediction work directly to implement block predictors, I did a systematic analysis of block prediction using various basic prediction components. This work tries to understand what types and combinations of predictors are effective at doing block prediction. In general, global and path predictors were found to be better than local and bimodal predictors at making predictions. However, combinations of two different types of predictors offer higher prediction accuracies than individual com-

ponents of the same size. Next, we analyzed multi-component hybrid predictors and partitioned the number of predictions made correctly by each individual component as well as combinations of components. Hybrid predictors with different types of components such as local, global, path, and bimodal have the potential to reduce the MPKI by more than 50%. We also found that hybrid predictors which include many global or path prediction components each using different lengths performed very well in an ideal scenario. This analysis indicated the potential of geometric history lengths in obtaining accurate predictions.

- Using the results of the systematic analysis, I propose hardware block prediction components. First, I evaluate several chooser predictors for a four-component multi-hybrid predictor and propose a one-of-four chooser predictor that uses the results of the component predictors. A state-less chooser achieves only 4.5% degradation in MPKI over a global-history based chooser while a component-outcome based chooser achieves a 4% improvement over the global-history based chooser. Second, I propose a mapping of the state-of-the-art branch predictors to exit prediction. The results for the OGEHL-like [60] exit predictor are disappointing due to chooser predictor inefficiencies. A scaled TAGE-like [62] exit predictor achieves over 12% reduction in MPKI compared to an equal-sized tournament predictor. For large 64 KB configurations, a multi-perceptron piecewise linear predictor using eight components performs very well. Third, I propose a general exit prediction technique called Post Prediction Encoding (PPE) that can use a good binary predictor as its component predictor. This predictor can directly use previously proposed branch direction predictors as its component predictors. A PPE predictor using piecewise linear branch predictor components achieves the lowest MPKI among the evaluated 16 KB predictors. Finally, I evaluate history-based indirect branch predictors and propose exit prediction-inspired indirect branch predictors as a component in the target predictor. Combining the best proposed components and design changes, I also show a 16 KB

improved block predictor that achieves 37.4% reduction in MPKI for the integer suite and 13.8% reduction in MPKI for the floating-point suite.

- There has been considerable work in understanding correlation and predictability of branches in programs. I have proposed a methodology to track correlation among blocks using perceptrons. Results show that more than 50% of the highly correlated branches are converted to predicates during hyperblock construction resulting in a significant loss in correlation. I also examine the relative importance of the local predictor component for an exit predictor compared to a branch predictor. The local predictor can effectively capture up to 6% of the highly correlated branches that the global predictor may not capture due to insufficient history length.

- The TFlex architecture consists of composable lightweight processors that require area and timing-efficient accurate predictors. These predictors are physically distributed across the small cores, but all the predictors from the cores participating in the execution of a thread work together to produce the block predictions for that thread. This requirement necessitates small, simple, and accurate distributed block predictors. I propose a classification scheme for predictors in such fully distributed architectures that classifies distributed predictors into four main categories called independent distributed prediction, banked distributed prediction, co-operative distributed prediction, and monolithic prediction. Though the distributed predictor designs are adapted from monolithic branch and block predictors, the challenges lie in making the distributed prediction efficient. I propose techniques to combine predictions from discrete components for multi-component predictors such as local/global predictors and local/TAGE exit predictors, ways to partition prediction tables, and a distributed return address predictor. The banked distributed predictor and co-operative distributed predictor achieve similar accuracies as an equal-sized monolithic predictor. I also show effect of different schemes on TFlex performance and compare the latency/accuracy trade-offs for different block ownership schemes.

1.5 Dissertation organization

This chapter provided a brief overview of distributed and block-based architectures, the importance of control flow speculation in such architectures, and the contributions of our work. The rest of this dissertation is organized as follows.

In Chapter 2 we discuss simple exit predictors inspired from branch predictors and present a brief summary of our results from the first infrastructure we used for TRIPS. This work led to the final design of the TRIPS prototype block predictor. We describe the design of the prototype predictor in detail including the organization of the components, functions of each component and predictor operations. We discuss implementation details such as timing and area constraints. Finally we report the predictor misprediction rates from the prototype chip and compare the rates with results from simulators.

Next, in Chapter 3, we first discuss the bottlenecks in the prototype predictor and how some of the bottlenecks become less severe when moving to a simple scaled version of the predictor. We present a detailed analysis of exit prediction in which we try to predict exits using individual components and combinations of components. This work helps in understanding the types of predictors that can predict exits effectively. We analyze target prediction independent of exit prediction and find the bottlenecks that can be fixed with simple design changes in the prototype target predictor. Finally, we examine the primary causes for a majority of the target mispredictions.

In Chapter 4 we use the results of our analysis and propose novel exit and target prediction components that can improve overall block prediction accuracy. We propose a mapping of branch predictors to block predictors, a prediction technique to use the best binary predictors in block predictors, a chooser predictor to extract the best out of component predictors and finally, an indirect branch predictor inspired by exit predictors. We compare the predictor results before and after applying these techniques to block prediction.

In Chapter 5 we move on from hardware techniques to identification of correlation loss due to block construction that leads to a lower prediction accuracy for exits compared

to branches. We propose a perceptron-based correlation analysis that tracks branches in hyperblocks and basic blocks and quantifies the loss in correlation among branches in hyperblocks compared to basic blocks.

We propose block predictors for TFlex in Chapter 6. We propose a classification scheme for block predictors in composable lightweight processors. We propose predictors inspired from monolithic predictors for each of the design points. We also identify the key bottlenecks in distributing a predictor and propose solutions for the same. We compare the predictors with a monolithic predictor and present the accuracy and timing trade-offs in distributed prediction design.

Chapter 7 discusses related work in block prediction, branch prediction, and distributed prediction. Finally, we provide conclusions, describe applicability of our work to other architectures, and discuss future directions in Chapter 8.

Chapter 2

Tournament Exit Predictor and Prototype Implementation

In this chapter, we first describe simple exit branch predictors. Next, we describe the TRIPS prototype predictor design in detail. We discuss the implementation of the predictor including area and timing trade-offs. Finally, we show the results of predictor evaluation on the TRIPS prototype, functional, and performance simulators.

TRIPS [56, 57] is a distributed, tile-based architecture that uses a block-based, block-atomic execution model. TRIPS uses control speculation to fetch and map successive blocks on to the execution substrate and employs data-flow execution within each block. The block-based and block-atomic properties of the architecture require effective block level control flow prediction, misprediction detection, and recovery. A TRIPS block [38], similar to a hyperblock [39] is a large single-entry multiple-exit block of instructions, some of which may be predicated [1]. Every branch in the block, called an exit branch, leads to transfer of control to the beginning of a new block. Though a TRIPS block can have multiple exits, exactly one of the exits fires during execution. Just as a branch predictor predicts one of two directions for a branch, an exit predictor guesses which one of several exits will be taken. By predicting an exit number *N*, we are, in effect, predicting a predicated path

through the block which leads to exit *N*. Only the path leading to this exit branch produces block outputs. The paths leading to the other exits do not produce block outputs. Since TRIPS is a distributed architecture, the predictor does not access the branches among the instructions fetched and dispatched to the execution tiles. Snooping the control network quickly for branch instructions is not feasible due to the distributed design. The Global Tile (GT) is physically away from the Execution Tiles (ET) to which instructions are dispatched directly from the caches. Hence, when a block is fetched, the predictor generally has only one piece of information about the block, namely the block address. Control information within the block, such as the number of branches, individual branch addresses, branch types, and targets are not available at the time of prediction. Hence the predictor also needs to predict some block and exit branch characteristics.

In the next few sections, we describe approaches to block prediction and basic exit predictor design. We discuss support for handling the constraints imposed on the predictor and describe the design and implementation of the prototype predictor.

2.1 Mechanisms for block prediction

Several mechanisms for branch prediction have been proposed over the last few decades. Using these previously proposed techniques, one can arrive at approaches to the block prediction problem. When a block is fetched, the predictor attempts to predict the address of the next block. There are three main ways to target this problem.

MBranch+Target

In the first approach, which we call *MBranch+Target*, multiple branch predictors are used to predict each exit branch in the block. The earliest branch in the block (usually, in program order) that is predicted taken is considered the predicted taken exit [11, 61, 80]. Target addresses corresponding to each of the branches in the block are predicted using a multi-component target predictor (either multiple target predictors or having multiple read

ports). Such a target predictor typically contains multiple components like a Branch Target Buffer (BTB) and Return Address Stack (RAS) [28]. Once the taken branch is predicted, the corresponding predicted target is chosen as the predicted next block address. The advantage with this approach is that, since we predict branches, we can likely get better prediction accuracy using the best branch predictors in the literature [24, 62]. Furthermore, branch prediction might be easier than block prediction since each predictor has to decide only between two options: taken and not-taken. One of the disadvantages with this approach is that predictor resources are allocated to each individual branch predictor and not every resource is always useful. For example, if the first branch is taken most often, the predictions from other structures are not useful. Further, this design is not very scalable if the number of exit branches is very high, say eight or more. Some proposals like the Multiple Block-Ahead predictor [61] use only as many bits as a typical branch predictor but the accuracy drops when predicting more than two or three basic blocks at a time.

Exit+Target

In the second approach, which we call *Exit+Target*, the predictor predicts the exit branch taken from the current block (by predicting an exit number). To predict the exit, exit history (comprising of a series of exit numbers called exit IDs) or path history (comprising of a series of block address bits representing the path taken to get to this block) can be used instead of typical branch history (sequence of taken/not-taken branch direction bits). Using the predicted exit and the current block address, we predict the next block address using a multi-component target predictor. Several previous high-bandwidth fetch techniques and decentralized/distributed processors have used this approach [16, 21].

DirectTarget

In the third approach, called *DirectTarget*, the predictor directly predicts the address of the next block using exit branch history or path history based on the block address [22, 47].

This design avoids predicting an exit first. Hence, it is simpler than the *Exit+Target* design. However, our early evaluations of this approach showed that the performance (in terms of misprediction rate) of this approach is inferior to the *Exit+Target* design. We do not consider this approach any further.

Since *Exit+Target* gives higher accuracies compared to *DirectTarget* and is more scalable compared to *MBranch+Target*, from now on, we use the *Exit+Target* block predictor design in all our evaluations. Our predictors always predict the exit first using branch history or exit history or path history and then predict the target using a target predictor. This design is implemented in the TRIPS prototype processor, described later in this chapter.

2.2 Tournament exit predictor

In an early version of the TRIPS processor design, a simple two-level block predictor similar to global branch predictors was used [44]. We then performed a systematic design-space exploration of hyperblock prediction [50]. We used the Trimaran compiler [76] infrastructure to generate hyperblocks that are mapped on to the TRIPS grid processor substrate [44, 56]. The goal was to arrive at a reasonably simple design for the exit predictor while allowing large fetch bandwidth and high exit prediction accuracy.

To build exit predictors we used two different kinds of histories: exit history, which is comprised of few lower order bits of exit IDs concatenated together, and path history, which is comprised of few lower bits of block addresses concatenated together. To make use of local correlation among exits within a block (self-correlation), we use local exit histories comprised of IDs of recently taken exits from the same hyperblock. Global exit histories are formed by a series of globally encountered recently taken exit IDs. A simple two-level global exit predictor and hyperblock are shown in Figure 2.1.

In the two-level local exit predictor the first level has the exit history table. The local exit history table is indexed using a hash of the block address and the local exit history for the current block is retrieved. The local exit history hashed with the block address is used

to index the second-level table (local prediction table) which contains the predicted exit ID and few bits for hysteresis (in our experiments, one bit was sufficient). The global two-level predictor is similar to the local predictor; only the first-level global history register differs. The global exit history register is a single register, containing recently encountered global exits, hashed with the block address and used to index the second-level global exit prediction table. Path history [45] is constructed using few bits from the addresses of recently executed blocks. The path history represents the global path taken by the program to reach the current hyperblock. In general, we found that exit history-based prediction is more accurate than path history-based prediction. We observed that one of the primary reasons for this difference is the heavier aliasing in the path predictor due to XOR-ing of several path addresses. On the other hand, when fewer path addresses were used, it proved less accurate due to less correlation captured. A similar problem is observed in path-based branch predictors [45].

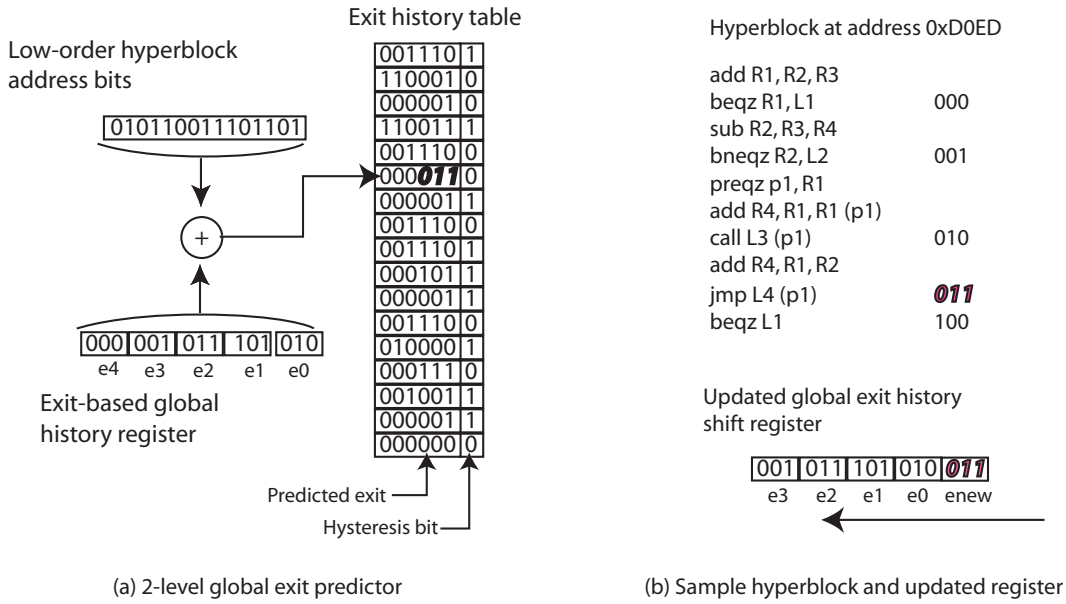


Figure 2.1: Two-level global exit predictor example

Design space exploration

We used a subset of SPEC 2000 and SPEC 95 benchmarks which we were able to compile successfully using the Trimaran compiler. We evaluated various combinations of local, global, and path-based exit predictors using the Trimaran compiler/TRIPS processor simulator infrastructure [44]. We compared the best exit predictors to branch predictors and multiple branch predictors. We give a brief summary of the relevant results below. These results guided us in the design of the TRIPS prototype predictor. Detailed results are reported in [50].

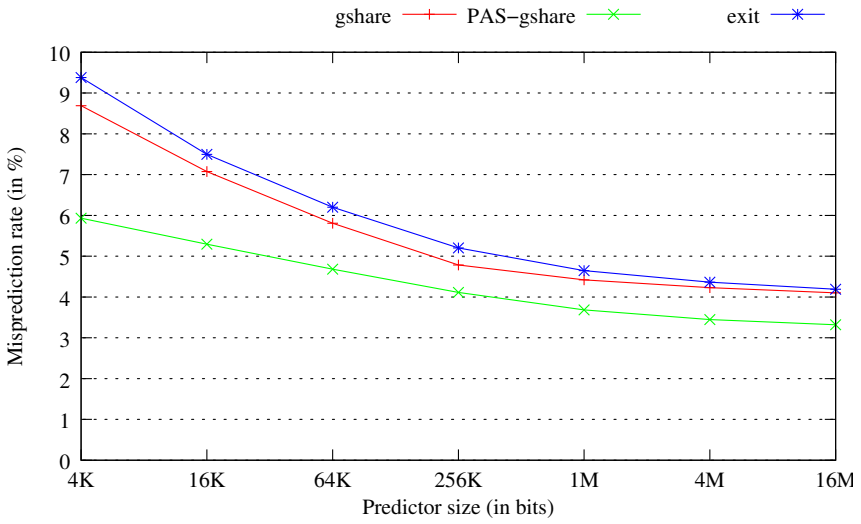


Figure 2.2: Comparison of misprediction rates of Gshare branch predictor, PAs/Gshare branch predictor, and tournament exit predictor for various sizes (4 KBits to 16 Mbits) for a set of SPEC95 and SPEC2K integer benchmarks.

- On the average, every block had 91 useful dynamic instructions. The average number of useful instructions for integer benchmarks was 37.
- The average number of exit branches in a block was 4.7 (4.4 for integer benchmarks).
- The average number of not-taken exits before encountering a taken exit is 2.6. Hence,

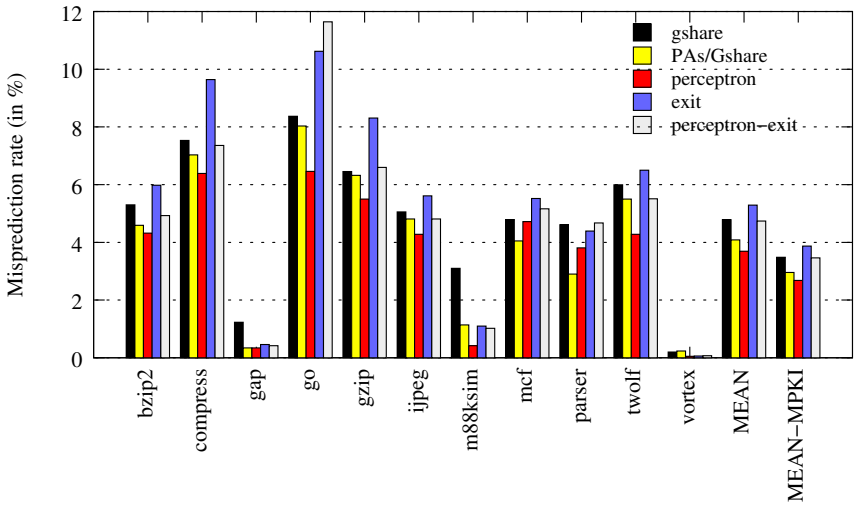


Figure 2.3: Comparison of misprediction rates of Gshare branch predictor, PAs/Gshare branch predictor, perceptron branch predictor, local/global tournament exit predictor, and perceptron exit predictor for a set of SPEC95 and SPEC2K integer benchmarks. All predictors are approximately 32 KB in size.

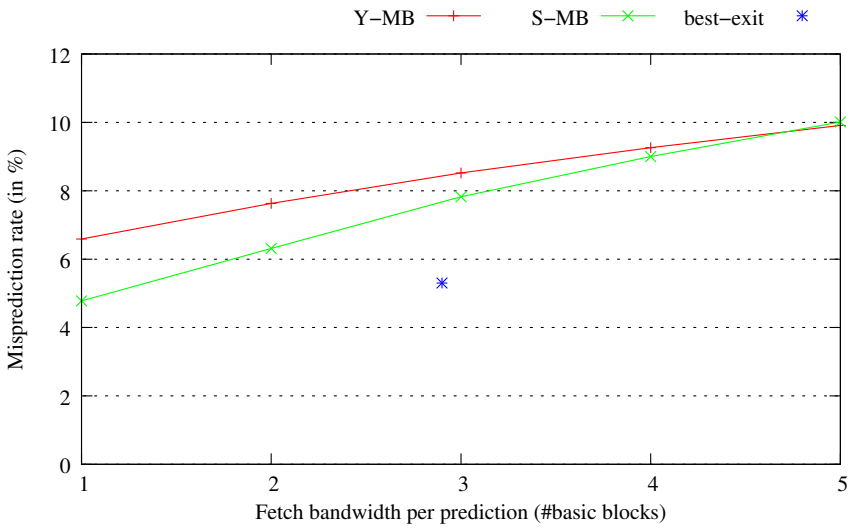


Figure 2.4: Comparison of misprediction rates of multiple branch predictor (Y-MB), multiple block-ahead branch predictor (S-MB), and tournament exit predictor for various fetch bandwidths for a set of SPEC integer benchmarks.

predicting the exit using an exit predictor instead of predicting each branch using a branch predictor eliminates 72% of the predictions.

- To construct the exit history, using between one and three lower order bits from the exit ID in the exit history instead of using the entire exit ID is sufficient. Lower order bits show the maximum variation among different exit IDs and this approach captures more exits in the history.

- Evaluations of various types of local, global, and path exit predictors (250 KBits) show that global predictors outperform path predictors and path predictors outperform local predictors.

- A local/global tournament [41] exit predictor (using a global history based chooser [4]) outperforms a global predictor of the same size. This predictor used different history lengths and exit ID lengths in its local and global components and was similar in structure to the Alpha 21264 branch predictor [29].

- The misprediction rates of exit predictors are significantly higher than that of similar history-based branch predictors. However, since the overall number of predictions to be made is much fewer and significantly more time can be taken to make each prediction, there is a lot of scope for improvement when going to more advanced designs. Figure 2.2 shows the misprediction rates of a tournament exit predictor in comparison with the Gshare [41] and PAs/Gshare [41, 82] branch predictors for various predictor sizes. All three predictors scale well up to 1 Mbits of storage. However, the gap between the exit and the branch predictors is significantly higher for predictors up to 64 KBits.

- The local/global tournament exit predictor was only slightly inferior to a mapping of the global perceptron predictor to exit prediction. Figure 2.3 shows the misprediction rates for three branch predictors (Gshare, PAs/Gshare, and global perceptron) and two exit predictors (tournament exit and global perceptron exit) for predictors of size

32 KB (256 KBits). Considering that the tournament exit predictor has a much simpler design compared to the perceptron exit predictor, it was chosen as the predictor for the TRIPS prototype.

- The misprediction rates of exit predictors are same or better than related multiple-branch prediction approaches when predicting more than two or three basic blocks at a time. Figure 2.4 compares the tournament exit predictor with Yeh et al's multiple branch predictor [80] and Seznec et al's multiple block-ahead predictor [61]. The multiple branch/block predictors achieve progressively higher misprediction rates as they are predictor higher number of basic blocks at a time. The exit predictor achieves the lowest misprediction rate of the three predictors while predicting an average of approximately three basic blocks for every hyperblock prediction.

- The local/global tournament predictor enabled significant performance improvement in the TRIPS processor as well as a VLIW processor using hyperblock compilation.

The TRIPS prototype predictor,described in the next section is a scaled-down version of the best local/global tournament predictor described briefly above (and in detail in [50]). This design was chosen for the relatively simple design (easier to verify in a prototype model) as well as reasonable prediction accuracies.

2.3 TRIPS block characteristics

A TRIPS block is a single-entry multiple-exit hyperblock-like region. It can have up to 128 useful instructions. The architecture imposes constraints on the number of loads, stores, reads, writes, and branches within a block. Every branch in the block is an exit branch (leads to transfer of control to the beginning of another block). Each branch instruction in the ISA has a three-bit space for the exit identifier. Hence there can be at most eight exit IDs in a block. However there can be more than eight exit branches in the block, some of them sharing exit IDs. For example, two exits one of which is frequently taken and another is

rarely taken can share the same exit ID. Similarly, exits going to the same target can share the same ID. There are no restrictions on the types of branch instructions in a block. A block can have any number of calls or returns or regular branches.

2.4 TRIPS prototype chip and predictor requirements

The TRIPS prototype [15, 57] is a 130 nm ASIC design with 170 million transistors and a maximum operating frequency of 366MHz. It has a tile-based design. The chip die photo with the various tiles marked is shown in Figure 2.5 and the block diagram of the TRIPS chip is shown in Figure 2.6.

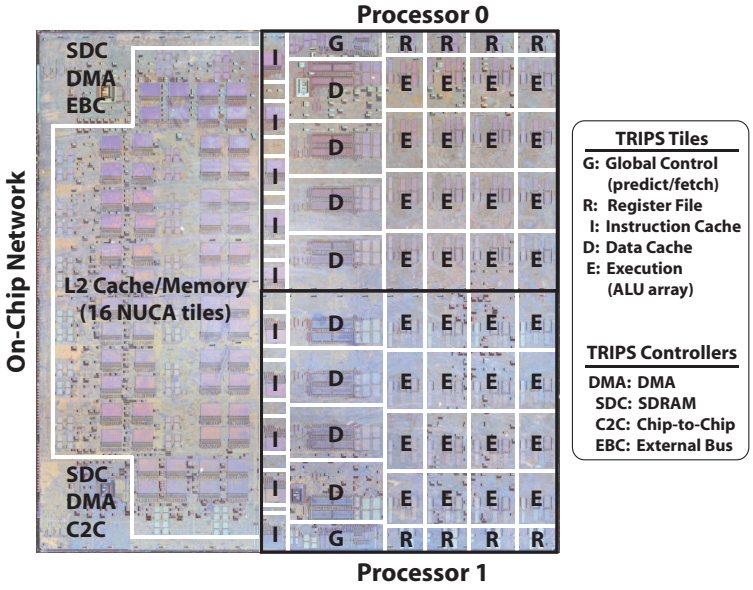


Figure 2.5: TRIPS prototype chip die photo with two 16-wide cores and 1 MB of L2 NUCA cache

The prototype has two 16-wide issue processor cores and 1 MB of L2 NUCA (Non-Uniform Cache Access) cache [32] on a chip. Each of the cores contains five major processor tiles: Global Control Tile (GT), Execution Tile (ET), Data Tile (DT), Instruction Tile (IT), and Register Tile (RT). The tiles are connected by several networks such as in-

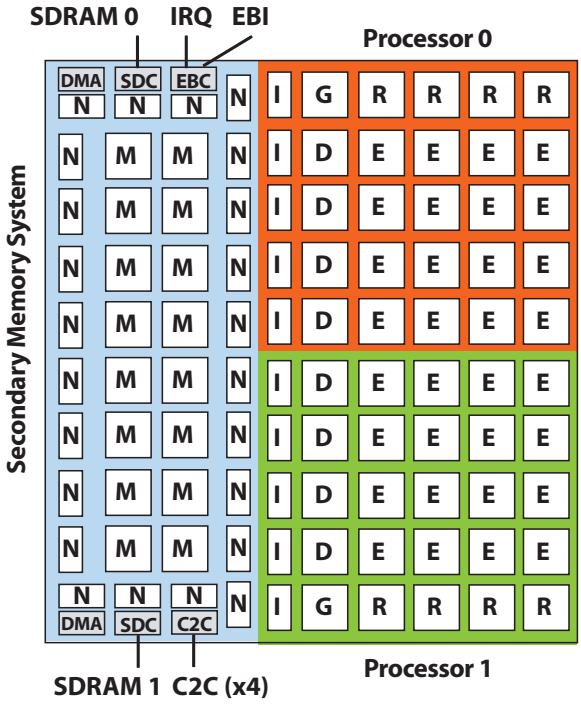


Figure 2.6: Tile-based design in the TRIPS prototype chip with two 16-wide cores and 1 MB of L2 NUCA cache

clude the Global Control Network (GCN), Operand Network for data (OPN), Global Register Network (GRN), and On-chip Network for L2 cache banks (OCN). The next block predictor is part of the GT. Each core contains one GT, five ITs, four DTs, four RTs, and 16 ETs. Every tile on the core also contains an OPN unit. The instruction window can hold 1024 instructions (equivalent to a eight blocks). In the steady state, the rate of fetch is one block every eight cycles.

The design requirements specified for the prototype predictor are listed below:

- Simple design in a small area (Global tile area budget was 2.0 mm^2). The complexity of the design would affect the verification time.
- The maximum number of exit IDs supported is 8, in any combination of branch

types (call/return/branch). Hence the exit numbers need not be more than three bits wide.

- The predictor should be able to predict call and return targets correctly in a block-atomic scenario.
- Highly accurate predictions are desired as the performance penalty of misspeculation is huge (1K entry instruction window).
- Reasonable prediction accuracy in SMT (Simultaneous Multi-threading) mode of operation.

Though the TRIPS block predictor has some design constraints not found in conventional branch predictors, block predictors have the following advantages compared to a branch predictor.

- The block prediction rate is lower than branch prediction rate (once every eight cycles compared to once every cycle or two).
- Implementing speculative updates for block predictors requires less storage since predictor histories and RAS pointers are only checkpointed every time a block is predicted (unlike every time a branch is predicted, in conventional processors).

2.5 Next block prediction in the TRIPS prototype processor

The design decisions for the prototype predictor had to be made based on results from the predictor designed in the Trimaran/TRIPS infrastructure described above. However, for the prototype, the software infrastructure was changed from the Trimaran compiler to the Scale compiler [68] targeting a newly specified TRIPS ISA. When a block predictor has to be implemented on real hardware, several issues arise depending on the Instruction Set Architecture (ISA), timing constraints, and area available for the predictor. The block

predictor's performance depends on the prediction algorithm, aliasing effects, as well as the global correlation property of the hyperblocks handed down by the compiler.

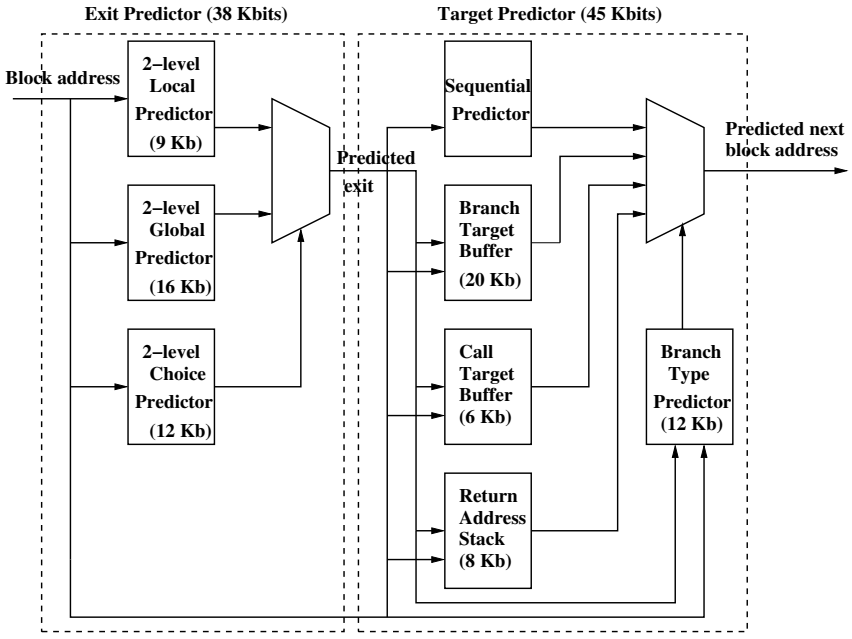


Figure 2.7: Major components in the TRIPS prototype predictor

The TRIPS prototype predictor includes a scaled-down version of the tournament exit predictor described in the previous section. It also includes a multi-component target predictor for predicting various types of branches. Based on the area constraints we arrived at a 5 KB exit predictor and a 5 KB target predictor. Since the next block predictor need only make predictions once per hyperblock, the number of predictions it would have to provide are far fewer than traditional branch predictors. Hence the predictor can take more time to make accurate predictions compared to a branch predictor. The prototype exit and target predictors are described below and shown in Figure 2.7. The individual table sizes are as marked. The predictor component breakdown is shown in Table 2.1. Details of predictor buffer sizes are shown in Table 2.2. The total size of all the major buffers is approximately 82.6 KBits. We describe the exit and the target predictors below in detail.

Next block predictor	Exit predictor	Local exit predictor	Local exit history table Local future file Exit prediction table
		Global exit predictor	Global exit history registers Global history file Exit prediction table
		Choice predictor	Global exit history registers Global history file Chooser table
	Target predictor	Branch type predictor	Branch type prediction table
		Branch target buffer	BTB table
		Call target buffer	CTB table
		Return address stack	Address stack pointers Link stack pointers History file Address stack Link stack

Table 2.1: TRIPS prototype block predictor - Component-wise breakdown of all storage structures

Table	Num. entries	Entry width (bits)	Total size (bits)	% total size
Local exit history table	512	10	5 K	6.1%
Local future file	8	19	152	0.2%
Local exit prediction table	1024	4	4 K	4.8%
Global exit history registers	4	12	48	0.1%
Global history file	8	12	96	0.1%
Global exit prediction table	4096	4	16 K	19.4%
Choice global exit history registers	4	12	48	0.1%
Choice global history file	8	12	96	0.1%
Choice chooser table	4096	3	12 K	14.5%
Branch type prediction table	4096	3	12 K	14.5%
BTB table	2048	10	20 K	24.2%
CTB table	128	44	5.5 K	6.7%
RAS address stack pointers	4	7	28	0.0%
RAS link stack pointers	4	7	28	0.0%
RAS history file	8	49	392	0.5%
RAS address stack	128	42	5.25 K	6.4%
RAS link stack	128	16	2 K	2.4%

Table 2.2: Next block predictor - Number of table entries, table entry widths, table sizes (in bits), and percentage of bits in each table

2.5.1 Prototype exit predictor

The exit predictor is a local/global tournament exit predictor with a global exit history based chooser. It is similar in structure to the Alpha 21264 tournament branch predictor [29]. The exit predictor uses approximately 38 KBits of storage. It has three components as shown in Figure 2.8.

Local exit predictor

The local exit predictor is similar to a two-level local branch predictor [81]. It has a 512-entry level-1 local (per-block) history table and a 1024-entry level-2 exit prediction table. Each entry of the local history table consists of a 10-bit per-block local exit history made up of the last five encountered taken exit IDs (truncated to last two bits - bits 0 and 1) for this block. Each entry of the local predictor table has a three-bit predicted exit and a hysteresis bit used for replacement. The level-1 table is indexed using the lower-order block address bits XOR-ed with the address-space ID (four-bit identifier of address space of the current thread). The local history read out from this table is XOR-ed along with the lower order block address bits (like Gshare [41]) to index into the prediction table.

Local branch predictors are not used in most of the branch predictors in recent microprocessors from the industry for several reasons. Local branch predictors take longer to access than global predictors as they access two tables sequentially compared to the single table access for global or path predictors. Also, implementing speculative updates with fix-up for local branch predictors is complex (time and area-consuming). Furthermore, local branch predictors show poorer accuracy when compared to global branch predictors. Due to these problems, designers have favored simple global predictors even though their accuracy is inferior to the accuracy of tournament predictors.

Despite these drawbacks found in local branch predictors, the local exit predictor is included as a component in the TRIPS prototype predictor because of the following reasons:

- Prediction timing is not crucial for block predictors, one prediction is made every

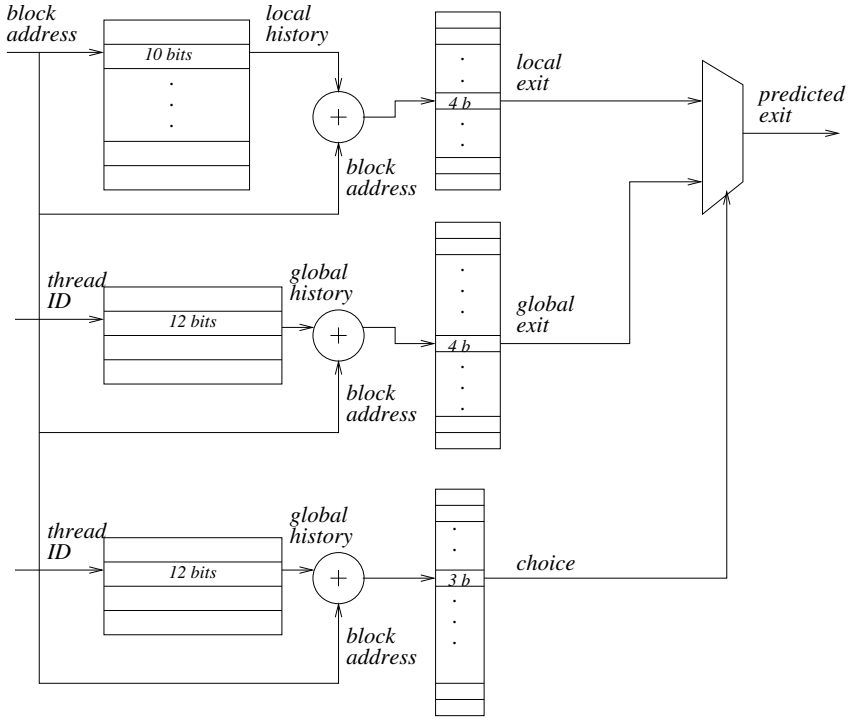


Figure 2.8: Local/Global tournament exit predictor with a global-history based chooser in TRIPS prototype predictor

eight cycles in steady-state.

- A local exit predictor, in general, has more potential to perform better than its branch predictor counterpart, since it also captures some global correlation if correlated branches are present inside the same block.

- Repairs after misspeculations are easier to implement at block granularities.

Global exit predictor

The global exit predictor is a two-level predictor which uses global history registers (GHRs) in the first level and a global prediction table in the second level. A GHR has a 12-bit history containing the last four global exit IDs encountered (full three-bit exit IDs included). Four

GHRs are maintained - one per thread, and the GHR chosen corresponds to the thread ID. The history is hashed with the block address [41] to index into the global prediction table. The prediction table contains 4096 entries with each entry storing the predicted exit and one hysteresis bit.

Choice predictor

The choice predictor (also called chooser) is similar to the chooser found in the Alpha 21264 predictor [29]. It uses global histories (12 bits containing the last six exit IDs, each ID containing two lower-order bits from the exit ID) and a second level table (containing 4096 entries, each three bits) of saturating counters. The counters select the global or local predictor depending on the most significant bit.

2.5.2 Prototype target predictor

The prototype target predictor predicts the next-block address using the current block address and the predicted exit from the exit predictor. Since exit branches can be of various types, support to predict various types of targets like branch target, call target, and return target is desired. Typically, modern target predictors have multiple components each tuned to predicting the target for one type of branch. The prototype target predictor has four components for predicting four types of branches. It also contains a branch type predictor to predict the type of the predicted exit branch. Predicting the final target from a multi-component target predictor requires knowledge of the branch type. Due to a distributed design, the prototype predictor does not see the branch instruction. This design necessitates predicting the branch type for the predicted exit branch.

The TRIPS prototype uses four different block sizes. Each block consists of a header chunk followed by one to four data chunks (containing instructions). Every chunk is exactly 128 bytes in length. Hence, a block can be 256 bytes or 384 bytes or 512 bytes or 640 bytes in length. The least significant seven bits of a 40-bit block address are zeroes.

The predictor control in the GT reads the block type (which indicates the number of chunks in the block) from the block header and sends the block type information along with other predictor inputs during predictions and updates.

In some processors such as the Alpha 21264 [29], target buffers such as the BTB are not used to store target addresses of direct branches/calls. Instead, the targets are quickly computed using an adder. This solution is not easily implementable in distributed architectures because the branch is not always fetched "close" to the predictor to make the offset read from the branch instruction quickly available to the predictor.

Another alternative, computing and storing the targets of all branches in the block header, is not scalable and uses header space. However Multiscalar [21] followed this approach to store four target addresses for every task. A TRIPS block can have more than eight exits even though the number of exit IDs allowed is only eight. Depending on whether the exit branch type is a direct branch or a direct call, each target may require up to 33 bits or approximately four bytes of storage in the header. To support eight targets with no restrictions on the branch types, we require 32 bytes of storage in the header chunk (which is one-fourth of the 128-byte header chunk). Using 25% or more of the header space to store branch targets is not feasible as the header needs to store block type information, read instructions, write instructions, and load-store IDs. If the header is made bigger to store the branch targets, the instruction cache utilization will be lower. Furthermore, the compiler may not know the indirect branch/call targets and return addresses at compile/link time. One alternative is to use the header to store only the direct branch targets as offsets (each target requiring fewer than 33 bits). In this case, we still require the CTB for call targets, the RAS for return targets, and a target buffer for indirect branches. Another disadvantage in storing the targets in the header is that the predictor and the fetch unit have to operate in lock-step mode and the predictor cannot run ahead of the fetch as it depends on the header chunk information fetched by the fetch unit for predicting the direct branch targets. Even though the current prototype does not have predictor run-ahead (which is useful for

I-cache prefetching [51, 52]), this is an important performance constraint for future TRIPS implementations. For the above reasons, we do not store targets in the header. Instead, we use target buffers in the hardware to predict various types of targets. We now describe the five different target predictor components below:

Branch type predictor (Btype)

The type of the branch can be learned by the predictor during completion time when the branch resolves and sends its resolved target, exit, and type information to the global tile. The prototype branch type predictor (Btype) predicts one of four types: sequential branch, branch, call, and return. The sequential branch type is learned internally to prevent sequential exit branches (whose target is the next hyperblock in program order) from occupying BTB entries. Using an adder, the target of sequential branches can be predicted and the BTB can be used only for non-sequential branches. Depending on the branch type predicted, one of four predicted targets (sequential, branch, call, return) is chosen as the predicted next-block address. The Btype predictor has 4096 entries, each entry containing a two-bit branch type and a one-bit hysteresis. The predictor is indexed using a hash (XOR) of the block address, predicted exit, and address space ID. For the target predictor buffers, we can either XOR or concatenate the block address bits and predicted exit to form the index. Our early evaluations showed that XOR-ing is better as the distribution in the table is more uniform in this case. XOR-ing is also especially useful when blocks have varying number of exits.

Sequential branch target predictor (Seq)

The sequential branch target is computed using a simple adder to find the address of the next block given the address of the current block and the current block size which is computed using the block type information sent as an input to the predictor.

Branch target buffer (BTB)

The BTB is indexed similar to the Btype predictor. It contains 2048 entries of target offsets. Each entry has a nine-bit offset and a one-bit hysteresis used for replacement. The branch target is computed by adding the (shifted) offset to the current block address. The offset is shifted left by seven bits since the block addresses are at 128 (or multiple) byte boundaries. Only targets for non-sequential branches are stored inside this table. Both direct and indirect branches are predicted using the BTB. Due to area and design time constraints we did not implement a separate indirect branch predictor. The drawbacks of using the BTB for indirect branches are the higher misprediction rates for indirect branches when compared to separate indirect branch predictors and the heavier aliasing in the BTB.

Call target buffer (CTB)

The CTB is indexed similar to the Btype predictor above. It contains 128 entries. Each entry has the absolute call target address (not including the lower-order seven bits which are always zeroes) and return address offsets. The call target is stored as an absolute address since call targets can be located far away from the calls (far calls in the virtual address space). The return offset corresponds to the return associated with the function that is called. The reason for the presence of the return target offset is explained below. Each entry has two hysteresis bits for replacement, one each associated with the call and return targets.

Call-Return mechanism and return address prediction

The call-return mechanism is somewhat different in TRIPS compared to conventional architectures that are instruction-atomic. To ensure block atomicity, function return points have to be at the start of a block since entering a block in the middle (after taking the call) violates the block-atomic constraints. Hence the caller block saves the return address (starting address of a different hyperblock) in a specific memory/register location and the callee fetches this information. Once the function is done and control is ready to return to the

callee function, the address is retrieved and the jump to the return target is done. Unlike conventional architectures, the return point need not be the next consecutive block after the callee block. It could be the beginning of any block within the calling function. There can be several calls in a block and each of these calls might have a different return address. Storing all the return addresses in the block header was not possible due to header size constraints, hence return addresses had to be learned dynamically. The mechanism used to learn return addresses when blocks with returns commit and using them later for prediction of return addresses is shown in Figure 6.10. The steps to store, retrieve, and learn return addresses are marked in the figure.

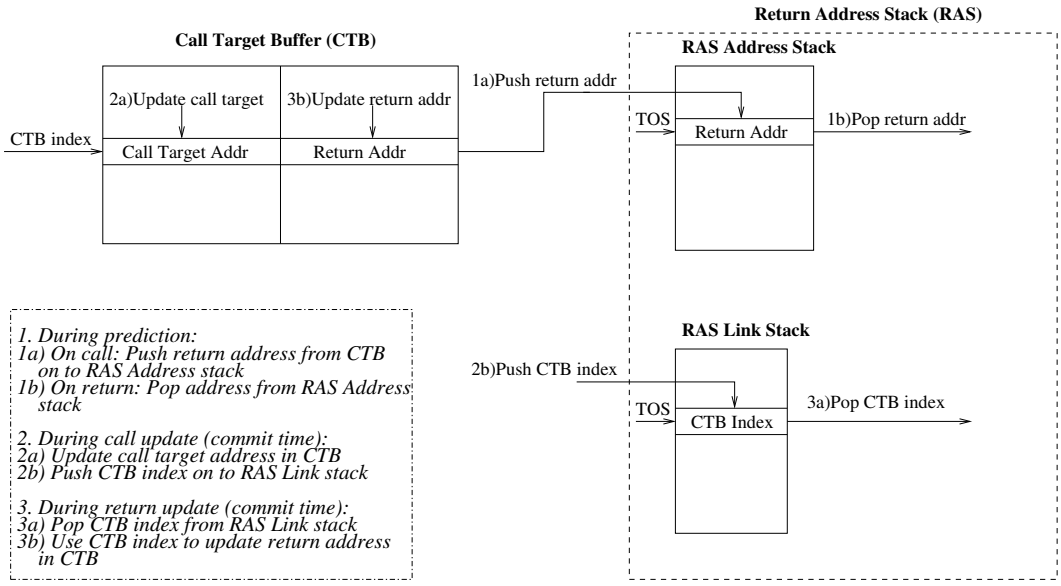


Figure 2.9: Call-Return mechanism in the TRIPS prototype predictor

The return address prediction mechanism involves five structures: the CTB, which holds the return addresses to be saved on to the Return Address Stack (RAS), the RAS which is the stack used to push and pop return addresses, the RAS History File (HF) which is used to save the top of the stack value and top of stack pointer to recover from mispredictions, a Link Stack (LS) to store the "link" between the call and the return (the CTB index),

and two sets of stack pointers (four in number, one per thread), one for the RAS and one for the LS. The RAS and the LS have 128 entries each. Each RAS entry contains the absolute return address (in base+offset form, due to timing constraints). Each LS entry contains the CTB index and call block address bits.

Whenever a call (block containing predicted call-exit) is fetched, it retrieves the return address corresponding to the called block from the same entry as the call target address. This address is then pushed on to the return address stack (RAS). When a return is fetched (block containing predicted return-exit) the stack is popped and the return address is sent to the Global Tile.

When a call is committed, the call target address is updated in the CTB and the index to the entry being updated is pushed onto the LS. When the corresponding return is committed, the entry is popped from the LS and the index (which is the link between the call and the return) is retrieved and used to update the return address in the CTB. Note that there is space to store only one return address corresponding to every call in the CTB. In case of function pointer-based calls, only the most recent return address can be stored in the CTB entry.

2.5.3 Support for Simultaneous Multi-threading (SMT) mode

Recall that a TRIPS chip has two 16-wide cores. Each TRIPS core has single-threaded and multi-threaded modes. The multi-threaded mode uses the simultaneous multithreading technique [46, 77, 79] to execute up to four threads. The next-block predictor can provide reasonable prediction accuracies in the SMT mode by making a small set of additions to the predictor. Note that the main goal of the predictor design is to optimize for single-threaded mode and support multi-threaded mode without significant overhead. Since the threads executing on the substrate could be from the same program or different programs, we use the thread address space ID to reduce aliasing in the predictor tables (instead of the thread ID, which can likely prevent constructive aliasing between threads from the same program).

Large structures are shared by the threads in SMT mode. To support effective predictions in the SMT mode, we added the following features to the predictor:

- The local history table index is generated by XOR-ing the block address and the thread address space ID, instead of just the block address, to achieve better utilization of the table.
- The global and choice history registers are maintained separately for each thread.
- The local, global, and choice predictor tables (2second level) are shared by all the threads.
- The indices for the Btype predictor, BTB, and CTB are generated by XORing the block address along with the concatenation of the address space ID and the predicted/resolved exit. Using the address space ID also in the index can help reduce destructive aliasing.
- Separate RAS and LS pointers are maintained for each thread.
- The RAS is used as a 128-entry deep single stack in single-threaded mode and used as four stacks, each 32-entry deep, in multi-threaded mode.

2.5.4 Predictor operations

In the TRIPS prototype, the next block predictor unit is responsible for making predictions, performing speculative updates, recovering from mispredictions, and updating the predictor tables at commit time. Each of these operations is detailed below.

Prediction

During prediction, the predictor receives the current block's information such as block address, dynamic block ID, block type (which denotes the block size), thread ID, and address-space ID. Using these, it predicts the exit taken out of the current block. The predicted exit

is used along with the current block address to predict the next block address. Since the GT does not need the exit information, only the next block address is sent as output from the predictor. At prediction time, the local, global, and choice predictor tables are read. The final exit is chosen using the chooser counter value from one of local and global values. The branch type, branch target address, and call target address are read from the respective tables using the block address and the predicted exit. The return address is read out from the top of the RAS address stack. The sequential target is also computed. Finally, using the predicted branch type, one of the four targets is chosen as the next block address.

Speculative updates

Speculative updates to the predictor structures are performed at the same time as the prediction operation. It has been shown in prior studies that recent history is important for predicting branches well [17, 27, 66, 67]. If the predictor history is speculatively updated, recovery must be initiated upon a misprediction to recover the history and other state to the state at the time of the mispredicted block. This is necessary to make accurate predictions along the redirected correct path of execution. When speculative updates to the predictor history All three histories used in the exit predictor (local history, global history, and global history in the chooser) are updated speculatively and recovered accurately. For accurate return predictions, the RAS stack is also pushed and popped during prediction time i.e., speculatively updated.

Speculative updates can be implemented for global histories using a global history file (GHF). The GHF is indexed using the dynamic block ID (which ranges from 0 to 7 for the TRIPS prototype). Whenever a new block is fetched it is assigned a new dynamic ID. Since the instruction window can hold a maximum of eight blocks, the dynamic IDs can be recycled after mapping eight blocks. Whenever an exit is predicted, the previous history from the global history register (GHR) is backed up in the GHF and the exit (or truncated exit) is shifted into the current GHR. When a block is mispredicted and the pipeline

flushed, the history corresponding to the mispredicted block from the GHF is written to the GHR [67]. This technique is used for the global histories in the global exit predictor and the chooser.

For using speculative local histories, a CAM (Content Addressable Memory) structure is maintained. This structure is called the local future file (LFF), which has eight entries (corresponding to a maximum of eight blocks in the instruction window) to store the eight latest local histories along with the block address bits as tag corresponding to each entry. Hence the local history table always stores non-speculative state and does not have to be repaired. On a block prediction, the local history table is indexed to retrieve the non-speculative history. Simultaneously, the LFF is searched (using the block address bits tag) for matches to retrieve more recent versions of the local history, if present. If a hit occurs in the LFF, the latest matching history is used otherwise the non-speculative history from the local history table is used.

In general, speculative updates for local predictors are not implemented in conventional superscalar processors due to the additional overhead of searching a CAM structure on the critical path to prediction. However, in TRIPS, a block-based execution model (with a maximum of eight blocks in the instruction window) is used that makes the number of searches drastically lower (eight searches) and the associative search much less complex. The local future file approach is similar to the technique described in [67].

For the RAS address stack, every time a block is predicted, the top of stack pointer and top of stack value are stored in the RAS history file (RAS HF). This history file is indexed using the dynamic block ID. This backup technique described in [66] has significantly less complexity than a full stack backup and can achieve over 99% restoration accuracy in practice.

Recovery from branch mispredictions

When a TRIPS block executes, exactly one of the exits will fire (taken exit) and send its target address to the Global tile. Once this exit resolves, the GT checks for a target misprediction. If the predicted target is different from the resolved target, the pipeline is flushed and branch misprediction recovery is initiated in the predictor. Note that resolution of exits from blocks may be out of program order. Repairs to predictor histories and the RAS address stack are done during the branch misprediction recovery operation.

To repair global and choice histories, we copy the backed-up histories from the global history files (GHF) of the global and choice predictor and write them into the global and choice history registers (GHRs). The GHFs are indexed with the dynamic block ID. After this, the current histories in the GHRs are updated with the correct resolved exit of the block that was mispredicted. For local histories, the local L1 table does not have to be updated since it is updated only at commit time. However, the history corresponding to the mispredicted block in the local future file (LFF) has to be fixed, i.e., the correctly resolved exit needs to be written to the history (instead of the predicted exit).

Among the RAS components, only the RAS address stack and RAS address stack pointers have to be fixed. The link stack components are not updated at prediction time. The RAS history file is indexed using the dynamic block ID and the backed-up address stack pointer and top of stack value are retrieved. The TOS pointer is first restored. Using this pointer, the TOS value is then restored in the RAS address stack. Further, if the resolved exit branch type is found to be a call, the CTB is read and the return address is pushed on to the stack. If the resolved branch type is a return, the stack is popped.

Recovery from load misspeculations

When a TRIPS block executes and a load-store dependence violation is detected, all blocks starting from the block that triggered the violation are flushed. The fetch engine then fetches blocks starting from the block that had the violation. Hence, in such cases, it is necessary to

restore the histories and stack pointers to the state before the violating block was predicted. This recovery is mostly similar to branch misprediction recovery as described above. However we do not apply changes corresponding to the violating block like shifting in the resolved exit or pushing/popping the stack based on resolved branch type. The key difference between recovery from load misspeculation and block mispredictions arises from the flush mechanism for when the misprediction is detected. When a load in block B is misspeculated, all blocks in the pipeline starting from the block B containing the violating load are flushed and the fetch engine refetches blocks starting from block B. However, when a block B has a branch misprediction, it means that the target address of that block is incorrect and so all blocks after block B are flushed. The fetch engine is then redirected to fetch from the correct path of execution starting from the correct target address of block B. Hence, during recovery from a load misprediction, the predictor state is recovered to the state before block B was fetched. During recovery from a branch misprediction, the predictor state is recovered to the state before block B was fetched (as in load misprediction) and then the correct operations corresponding to the block B are performed.

Commit-time (non-speculative) update

On completion of the oldest block in the instruction window, the fetch engine requests the predictor to perform commit-time updates using the resolved branch address, resolved exit, and branch type of the resolved exit. The local predictor table is indexed using the local history from the local history table and updated with the resolved exit (taking care of replacement using the hysteresis bit). Then the local history is updated. For the global and choice predictors, the predictor tables are indexed using the histories from the GHFs (which corresponds to the correct history at the time of prediction). The global predictor table is updated with the resolved exit. The chooser predictor counter is incremented or decremented depending on whether global or local predictor makes the correct prediction. If both or neither give the correct prediction, then the chooser counter is not updated.

For the target components like Btype, BTB, and CTB, updates are made while using hysteresis for replacement (BTB and CTB are updated only if the resolved branch type is relevant). If the resolved exit branch type is a call, the index to the CTB is pushed on the RAS link stack (along with the call block address bits). If the resolved branch type is a return, the CTB index is popped off the RAS link stack and the CTB entry is updated with the resolved return offset (calculated as the difference between the call block address and the resolved return address).

During commit time updates, the latest table entry value is compared with the resolved value for each table, instead of storing the predicted value and then comparing it with the resolved value. Hence for all the tables that use hysteresis-based update, the corresponding entries must be read out first to be compared with the resolved values and then written to based on two constraints: whether there was a mismatch in the current entry value and resolved values and the hysteresis bit value.

2.5.5 Predictor implementation and verification

The next block predictor was first designed using the results of the exit predictor design space exploration [50]. The predictor was scaled down to fit the area requirements of the Global Tile (2.0 mm^2). The design was tuned based on timing and area constraints. We first implemented the predictor in a performance simulator and a branch prediction trace simulator. After testing the predictor, the predictor was implemented in RTL using Verilog. Following this functional testing and performance verification of the predictor were completed.

Predictor area

The predictor area was limited by the GT area (2.0 mm^2). The GT includes the predictor, I-cache directory, some small tables, and control logic. The predictor takes up about 70% of the area of the GT. Most of the area in the predictor is used by the various tables. Due to area

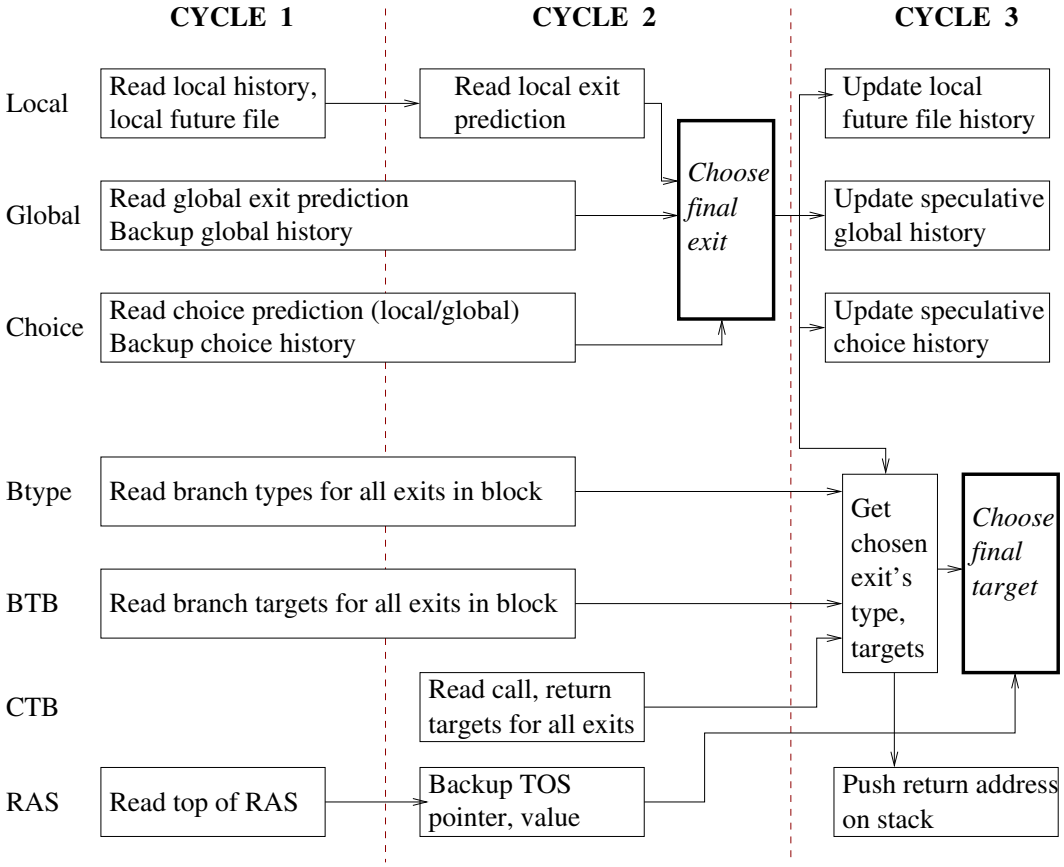


Figure 2.10: Prediction and speculative update high-level timing diagram showing exit and target prediction

restrictions, the predictor tables were kept small wherever possible. SRAMs occupied less area than register files but SRAMs had approximately twice the access latency compared to register files. Hence, register files were used for tables that needed to be accessed quickly like the local history table, local predictor table, and CTB table.

Predictor timing

Based on the instruction fetch bandwidth and steady state prediction rate, it was determined that a prediction and an update should be completed in no more than eight cycles. This latency is significantly higher than in conventional processors using branch prediction. Due

to the long latency allowed for making a prediction (up to four cycles), the predictor can use a simple blocking design instead of complex pipelined design. We show a high-level overview of prediction timing in Figure 2.10. This figure summarizes the prediction and speculative update operations happening in each cycle.

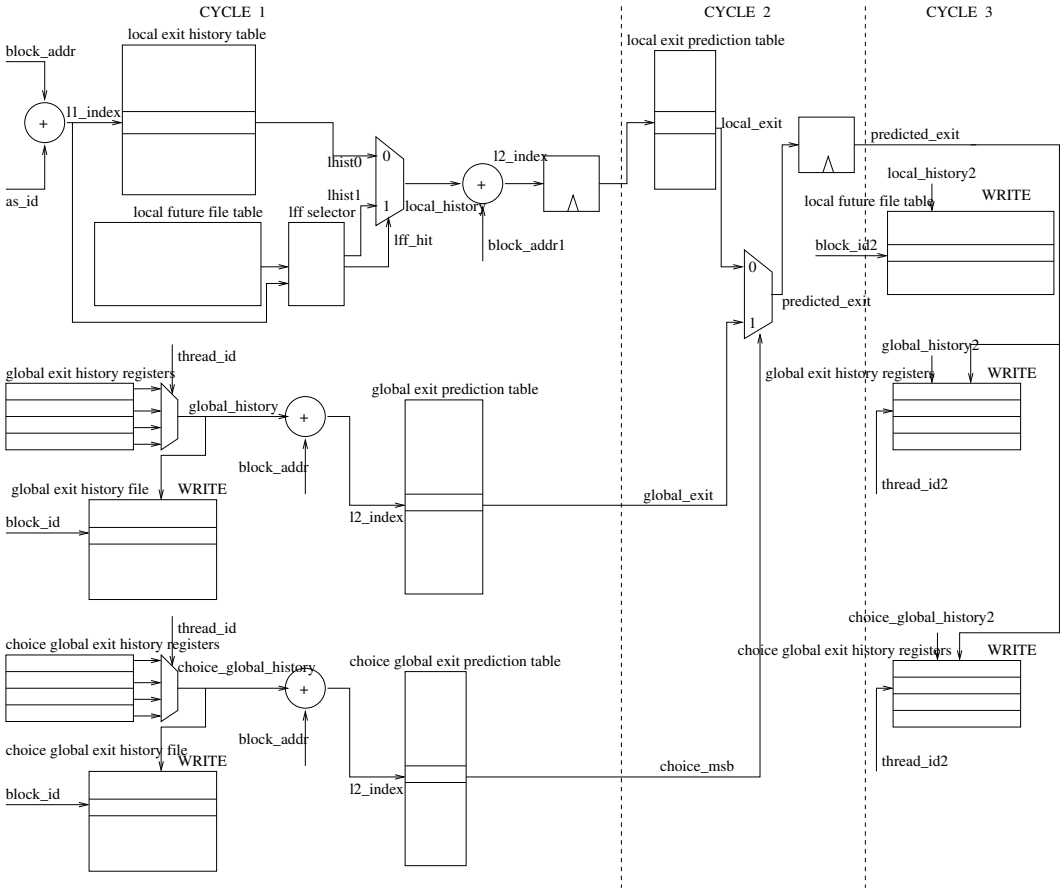


Figure 2.11: Prediction and speculative update low-level timing diagram showing exit prediction

Detailed prediction/speculative update operations are shown in Figures 2.11 and 2.12 respectively for the exit and target prediction components. These figures show the latches read and tables accessed in each cycle, mux selection, and addition of offsets to generate targets. The exit prediction latency required slightly over two cycles (due to sequential access of the two tables in the local predictor as shown in Figure 2.11).

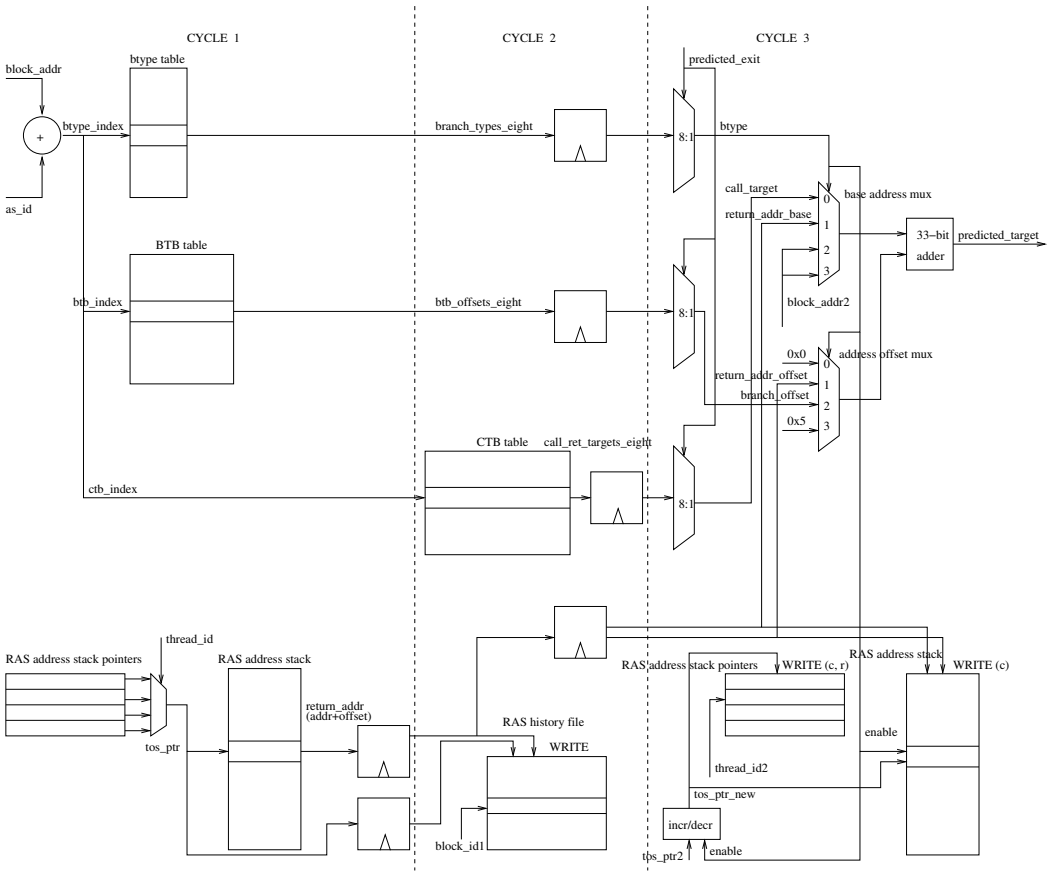


Figure 2.12: Prediction and speculative update low-level timing diagram showing target prediction

To save time, the target predictor tables are read in parallel to the exit predictor tables. The branch types and various targets for all eight exit IDs in the current block are read out from the tables as in Figure 2.12. Then, at the end of the exit prediction (beginning of the third cycle), we use multiplexers to choose the branch type, branch target, and call target based on the predicted exit. Finally we choose the correct target based on the predicted branch type. The predicted target is ready to be latched before the end of the third cycle. In parallel with the prediction, backup of histories, top of stack, and speculative updates are performed. The global and choice histories are backed up in the first cycle and speculatively updated after the exit is predicted in the third cycle. The RAS pointer and top of

stack value are backed up in the first cycle. If the predicted branch type is a call, the stack push operation happens in the third cycle. Thus prediction and speculative updates together take three cycles.

Updates to the predictor also take three cycles. The predictor is implemented as a blocking predictor that takes three cycles for each operation (even though internally, branch misprediction and load misspeculation updates may require less time depending on the resolved branch type). This design presents a simple interface to the GT instead of specifying different latencies for each operation. It also satisfies the prediction requirement of making one prediction and one update in eight cycles or less, in the steady state.

The table access times and cyclic dependences between call and return updates could have made pipelining difficult but we chose not to pipeline the predictor. Another option could have been multiple ports, but area restrictions made us choose single-ported structures. Implementing the predictor with blocking access using single-ported structures was possible only due to the large slack allowed between predictions.

Predictor verification

Functional and performance verification of the predictor was completed using hand-coded micro-benchmarks that tested individual components in the predictor. For example a small microbenchmark implementing a chain of calls could test how effectively the CTB and RAS perform. Rigorous testing of the predictor and functioning within the GT was completed using a separate detailed GT testbench and several other real-world benchmarks.

2.6 Predictor evaluation

In this section, we first briefly describe our evaluation methodology and then present results from the prototype and simulators.

2.6.1 Evaluation methodology

We use the Scale compiler [68] infrastructure to generate hyperblocks (highest level of optimization with *-Omax*, inlining, and call and return optimization). The *-Omax* optimization enables the highest level of optimization in the Scale compiler which includes hyperblock construction and predication. We use the SPEC2000 benchmark suite for evaluation. We were able to compile and checkpoint ten of the integer benchmarks and nine of the floating point (FP) benchmarks successfully. We do not evaluate *gap* due to compilation problems and *sixtrack* due to failure in our simulation-region finding infrastructure. The remaining benchmarks from the suite are not supported by our compiler (C++, Fortran90). These 19 benchmarks are highly representative of the entire benchmark suite containing 26 benchmarks and use these benchmarks throughout this dissertation.

Simulation phases

For the evaluation on the TRIPS prototype, we run the SPEC benchmarks to completion using reference inputs. For all other experiments, we use the Simpoint tool [63] to find representative regions in Alpha 21264 [29] binaries with a length of 100 million instructions. We used the Alpha binaries as they are representative of a typical RISC ISA and the standard simulation length of 100M instructions is a more accurate representation of the fraction of a benchmark executed than counting blocks or instructions (including dataflow overheads) in the TRIPS ISA. Once, the regions are found, we then translate the regions specified in terms of instruction count boundaries to calls and returns (as accurately as possible). This semi-automatic process results in each benchmark's representative region being bounded by *M* number of calls to a function X and *N* number of returns to a function Y. To mark the boundaries using calls and returns with well-defined characteristics, we may increase the Simpoint region size until we get proper boundaries. This translation to calls and returns is done so that frequent change in binaries (resulting from frequent improvements to the compiler) does not result in changing the Simpoint region boundaries.

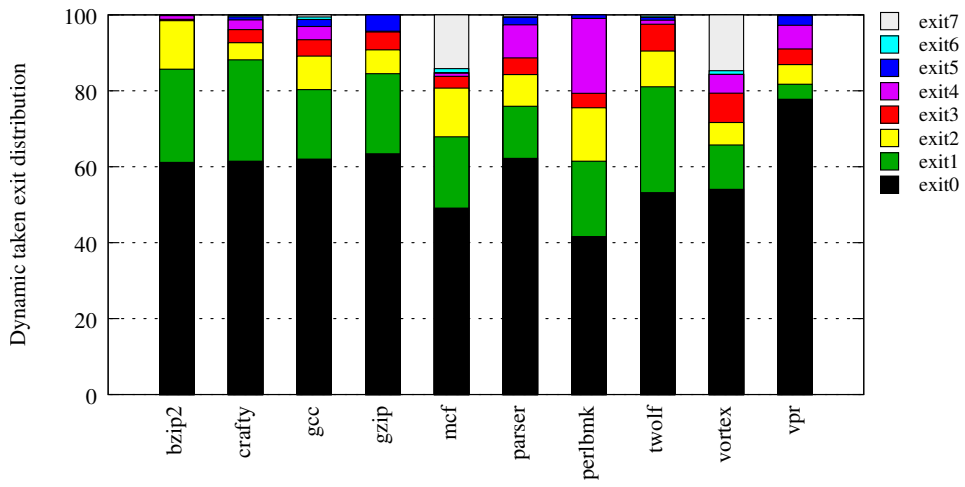
SPEC Int	bzip2	crafty	gcc	gzip	mcf	parser	perlbmk	twolf	vortex	vpr
# unique functions	4	32	937	17	5	179	42	22	243	23
# unique blocks	100	660	14094	154	53	1724	210	177	1720	227
# useful insts (M)	118.7	113.2	98.6	115.5	91.3	523.7	109.4	95.8	149.0	90.0
SPEC FP	ammp	applu	apsi	art	equake	mesa	mgrid	swim	wupwise	
# unique functions	23	2	4	5	5	53	45	3	12	
# unique blocks	260	36	44	73	59	350	371	26	126	
# useful insts (M)	444.7	279.7	573.6	191.6	342.5	150.2	196.6	389.5	103.7	

Table 2.3: Unique function, unique block, and instruction counts for SPEC2K integer and FP benchmarks in the simulated Simpoint region

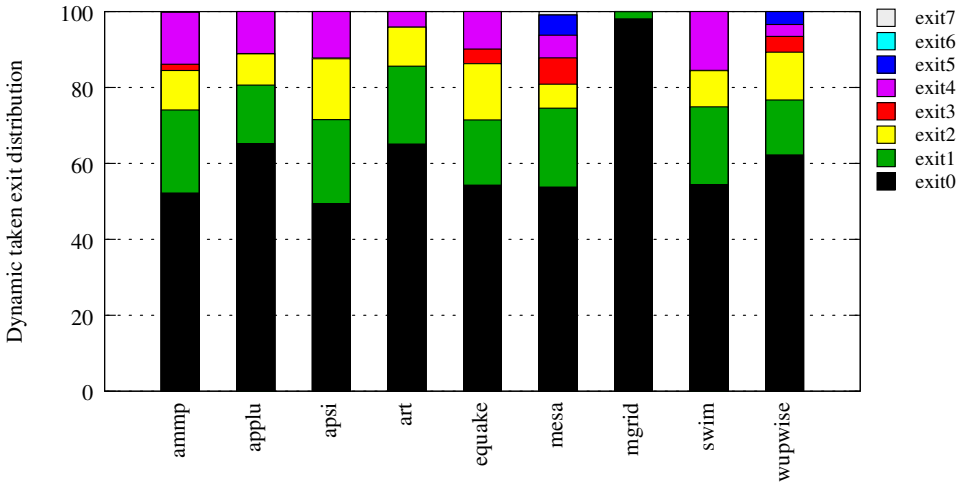
Table 2.3 shows the number of unique functions, blocks, and total number of useful instructions executed (moves and mispredicated path instructions eliminated) for each benchmark in the simulation region. Results are presented for both integer and floating point benchmarks. There is significant variation in the number of functions, blocks, and instructions among the different benchmarks. The highest number of unique functions are found in *gcc*, *parser*, and *vortex*. The presence of large number of unique functions poses a challenge in the CTB design.

The number of unique blocks is reasonably high in several of the benchmarks. However, some FP benchmarks and *mcf* have few unique blocks. The presence of a large number of unique blocks poses a challenge in the exit prediction components as well as in the BTB and Btype tables. The number of FP instructions in TRIPS is much higher than equivalent RISC code from the Alpha because the absence of a hardware floating-point divider in the TRIPS prototype necessitates software emulation of the division. For the integer benchmarks however, except for parser which has very small blocks, most of the instruction counts are much closer to the Alpha counts.

Figure 2.13 shows the dynamic taken exit ID distribution for the benchmarks in the



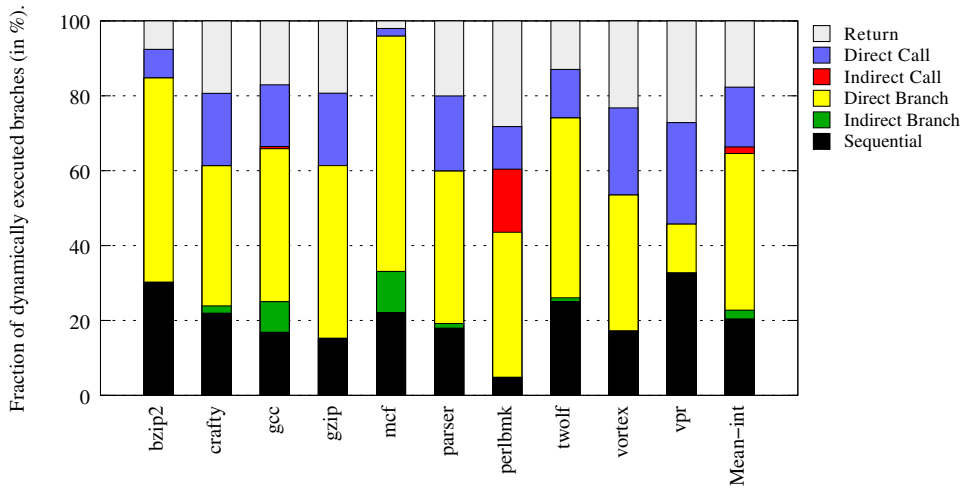
(a) SPEC2K integer benchmarks



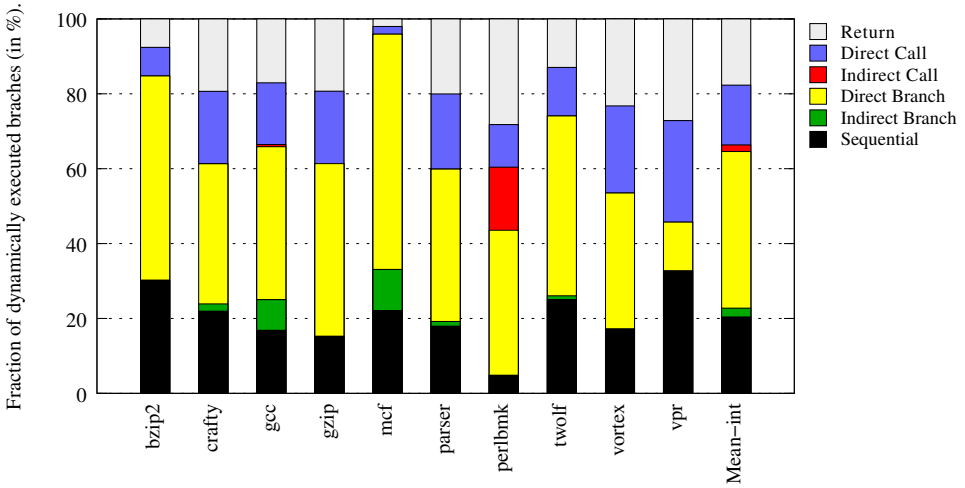
(b) SPEC2K FP benchmarks

Figure 2.13: Dynamic distribution of taken exit IDs for SPEC integer and floating-point benchmarks

simulation region. We find that exits with IDs 0 through 3 are the most frequently taken while the remaining exit IDs are seen less often. However most of the benchmarks have some taken exit IDs ranging from 4 through 7, even though their percentage is low.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 2.14: Dynamic distribution of taken branch types for SPEC integer and floating-point benchmarks

Figure 2.14 shows the distribution of different branch types of taken exit branches in the simulation region. Branches are classified into six categories based on their dynamic

behavior. Sequential branches have the next sequential block in program order as their target. Direct branches include regular direct branches (target computed from offset in branch instruction) as well as indirect branches which have only one target dynamically.

The category for indirect branches includes indirect branches that have multiple targets observed dynamically. Calls are split into regular calls and indirect calls (with multiple dynamically seen targets). The last category is for returns. We notice that the majority of the benchmarks are dominated by sequential branches and other direct branches. However for some benchmarks like *gcc* and *mcf* there is a significant percentage of indirect branches. For *perlbmk* we see a high number of indirect calls. There are virtually no indirect branches or calls in the FP benchmarks.

Simulators

Other than reporting predictor results from the TRIPS prototype, we use two simulators for our evaluation in this chapter and in the rest of this dissertation. The first is a branch prediction simulator built from the TRIPS functional simulator. This simulator is used for the predictor analysis and to report branch prediction results. We evaluate new predictors and improvements to existing predictors using this simulator. The second is the TRIPS performance simulator which models the prototype to within 11% accuracy. This simulator is used to report performance improvements and effect of speculative updates on the predictor whenever required.

Evaluation metrics

We use the execution time and speedup metrics when showing performance results. To measure the effectiveness of a predictor we use the following two metrics.

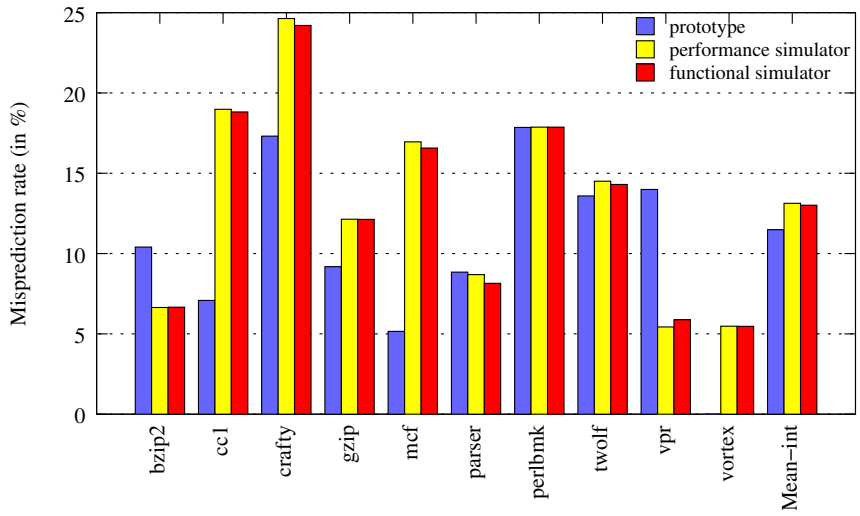
- Misprediction Rate (reported in %): This is the percentage of total predictions that were mispredicted. This measures the absolute effectiveness of the predictor.

- MPKI: Mispredictions per Kilo Instructions (MPKI) is a metric that measures the number of mispredictions seen per 1000 useful committed instructions. This metric is useful as a direct indicator of the processor performance. A lower MPKI value results in fewer pipeline flushes due to branch misspeculations, better utilization of the instruction window and higher performance. We count only the true useful instructions as given in the table.

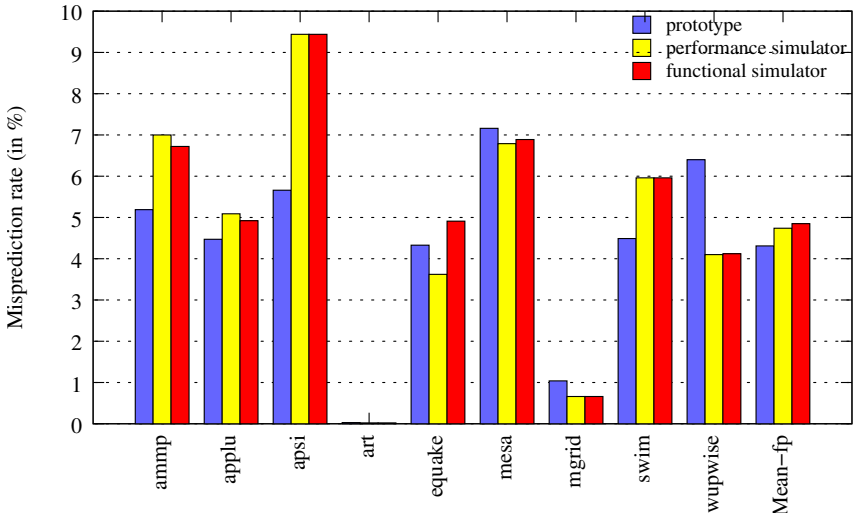
2.6.2 Predictor results

Figure 2.15 shows the misprediction rates of various SPEC benchmarks (integer and FP separately). The first bar for each benchmark shows the misprediction rate from the prototype, the second bar shows the misprediction rate from the TRIPS cycle-accurate performance simulator, and the third bar shows the misprediction rate from the functional branch predictor simulator. The branch misprediction rate from the prototype was determined by using the total number of mispredicted blocks and the total number of committed blocks. These values are available from hardware performance counters. When the program ends, the hardware performance counters are dumped to a file and the branch misprediction rate is calculated. All benchmarks were run to completion (with reference inputs) on the prototype. Since there were some problems running vortex, we do not show results for vortex from the prototype. For the performance and functional simulators, we show results for Simpoint [63] regions (equivalent to approximately 100M RISC instructions) as stated above.

The misprediction rates for the SPEC integer suite are 11.49%, 13.13%, and 13.01% respectively and the rates for the FP suite are 4.31%, 4.74%, and 4.85% respectively for the prototype, performance, and functional simulators. The mean misprediction rates for the block predictor are much higher compared to conventional branch predictors (which achieve misprediction rates of 5% or less for integer benchmarks). Potential sources of high misprediction rates can include inefficiencies in the predictor design, using a simple



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 2.15: Prototype predictor misprediction rates for SPEC integer and floating-point benchmarks from the hardware prototype, performance simulator, and branch predictor functional simulator

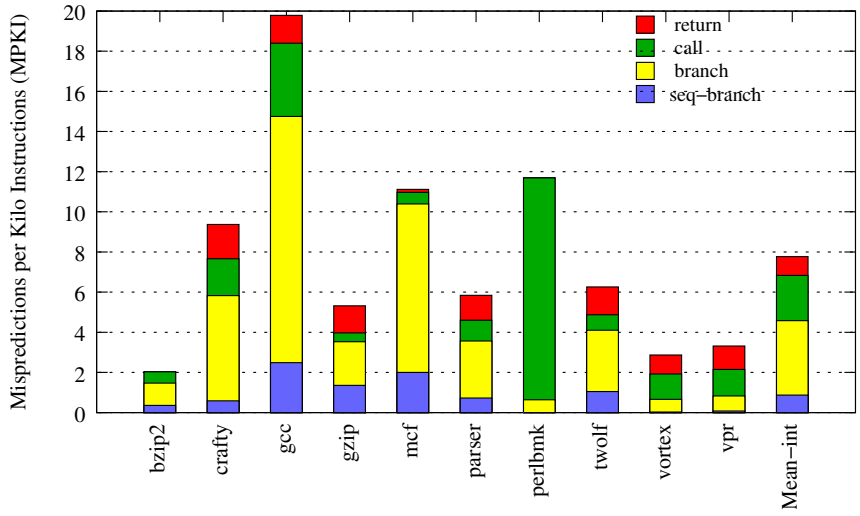
predictor design or inherent nature of blocks.

The Simpoint regions give a good approximation of the entire benchmark suite when considering misprediction rates for most of the benchmarks. Among integer benchmarks *crafty* and *vpr* show significant difference in the misprediction rates for the entire benchmark (from the prototype) when compared to the simpoint region (from the simulators). In the FP suite, *ammp*, *apsi*, *swim*, and *wupwise* show marked difference between the misprediction rates for the entire benchmark and those for the simpoint region. One of the reasons for this difference is that we use single Simpoint regions instead of multiple Simpoint regions to evaluate each benchmark. A single Simpoint region can cover at most two program phases and may not effectively capture the behavior of the entire program for some benchmarks.

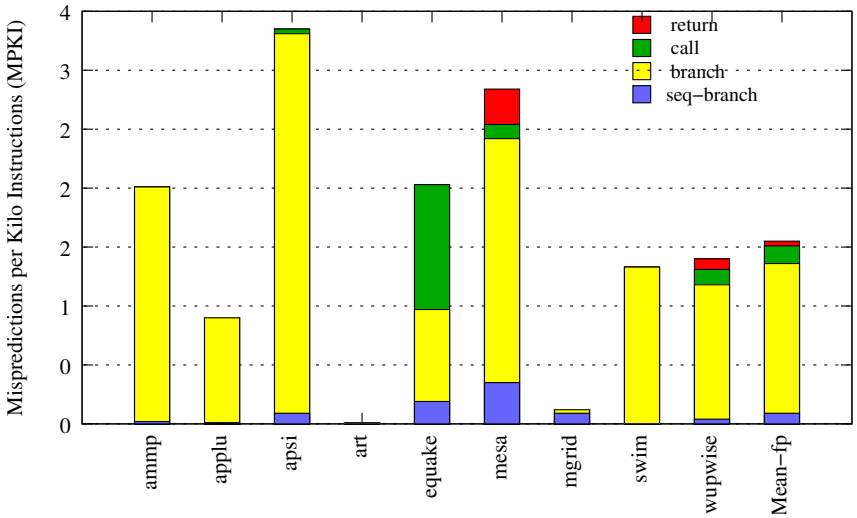
The difference in misprediction rates from the cycle-accurate performance simulator and functional simulator is negligible. The reason for the small difference is that speculative updates are implemented for all histories as well as the RAS. This speculative update mimics the immediate update of predictor state done in the functional model.

MPKI breakdown by branch type

Figure 2.16 shows the MPKI breakdown from the branch predictor functional simulator. The breakdown is shown by branch type. The total integer MPKI is 7.77 and floating-point MPKI is 1.55. The individual breakdown for integer is 0.87 from sequential branches, 3.71 from other branches, 2.25 from calls, and 0.94 from returns. For the FP suite, the breakdown is 0.09 from sequential branches, 1.27 from other branches, 0.15 from calls, and 0.04 from returns. Most of the mispredictions are branch or call mispredictions. However this does not directly point to an inefficiency in the BTB or the CTB. Component-wise breakdown presented in the next chapter will show the inefficiencies in the predictor.

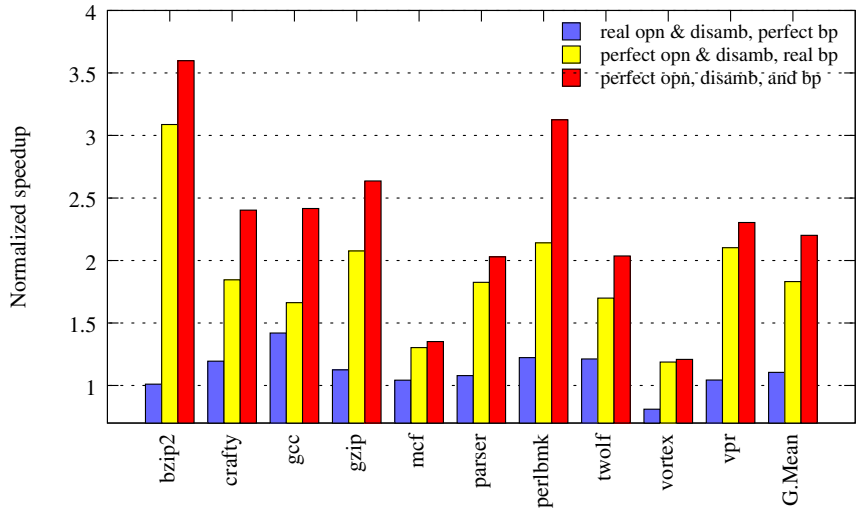


(a) SPEC2K integer benchmarks

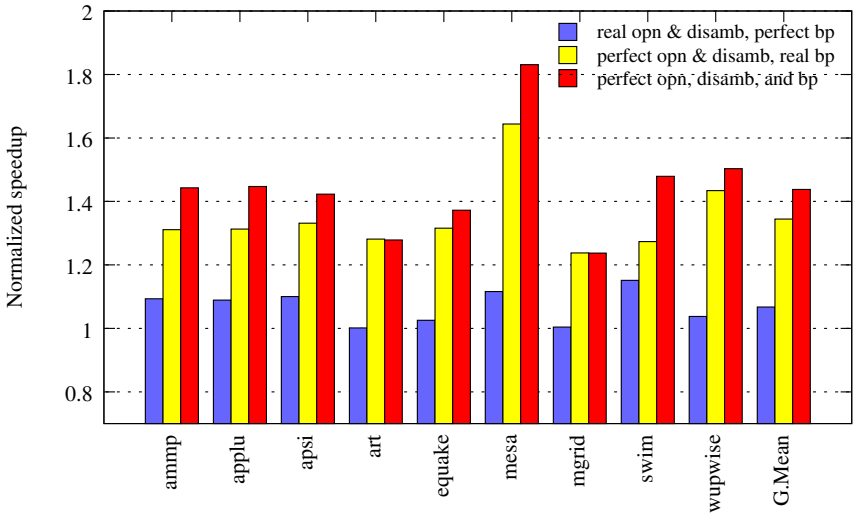


(b) SPEC2K FP benchmarks

Figure 2.16: Prototype predictor MPKI breakdown by branch type (sequential branch, branch, call, and return) from branch predictor functional simulator



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 2.17: Comparison of speedups for three TRIPS configurations with respect to a baseline realistic TRIPS prototype configuration (normalized speedup of 1) for SPEC integer and FP benchmarks. The first bar represents the relative speedup achieved by a perfect block predictor. The second bar represents the relative speedup achieved by a perfect OPN, perfect memory disambiguation, and realistic block predictor configuration. The final bar represents the relative speedup achieved by a perfect OPN, perfect memory disambiguation, and perfect block predictor configuration. Geometric mean of speedups is also shown.

Potential for better performance by improving block prediction

Figure 2.17 shows the relative speedups achieved by different TRIPS configurations over a baseline realistic configuration. The evaluations are done using the TRIPS performance cycle-accurate simulator. The baseline models the TRIPS prototype processor realistic configuration. The first bar for each benchmark shows the speedup obtained by perfect block prediction. The mean speedup for integer benchmarks is 10.6% and the speedup for floating-point benchmarks is 6.7%. This is the maximum speedup that can be obtained by improving the block predictor alone in a realistic configuration.

The second bar represents a configuration with perfect operand network communication (OPN) and perfect memory disambiguation with the realistic 10 KB TRIPS prototype predictor. Only the OPN and disambiguation is made perfect. The memory latency and other factors are kept realistic. The speedup obtained in this configuration is 83.1% for integer benchmarks and 34.4% for FP benchmarks. These numbers indicate the potential for significant improvement in the operand network communication and memory disambiguation mechanisms. We simulate this configuration to remove the major bottlenecks in TRIPS to isolate the performance impact of block mispredictions. The third bar shows the speedup with perfect OPN, perfect disambiguation, and perfect block prediction. The speedup over the baseline is 120.2% for integer programs and 43.7% for FP programs. When comparing the second and third bars, we find that, on average, when the OPN and disambiguation are perfect, block mispredictions account for 20.2% and 6.9% of the performance loss for integer and FP benchmarks respectively.

2.7 Summary

This chapter presented an introduction to block prediction in TRIPS and results from our design space exploration with the Trimaran/TRIPS simulator infrastructure which led to the exit predictor chosen for the TRIPS prototype. Next, we presented the design and im-

plementation of the TRIPS prototype next block predictor. We evaluated the prototype predictor using SPEC2K benchmarks on the prototype chip, performance simulator, and functional simulators. We also showed the performance loss due to block mispredictions. The prototype predictor has a two-stage prediction mechanism (exit prediction followed by target prediction). The exit predictor is a scaled-down version of the best simple exit predictor from our design space exploration experiments. The prototype predictor uses about 10 KB of storage, occupies about 1.4 mm^2 area and uses a simple three-cycle blocking design. It has support for branch type prediction, block-atomic call-return semantics, and SMT mode. Our measurements from the prototype chip indicate that the predictor achieves 11.5% misprediction rate for the integer suite and 4.3% for the floating-point suite.

The next chapter will present the inefficiencies in the TRIPS prototype predictor and an analysis of block predictors to understand the predictors that are suited for block prediction.

Chapter 3

Analysis of Block Predictors

In this chapter, we first present the inefficiencies in the TRIPS prototype predictor that led to a high misprediction rate compared to conventional branch predictors. We then determine how exit prediction can be made by looking at the performance of basic address and history-based prediction components. This retrospective analysis will provide guidelines to improve block predictors. There have been several past studies for understanding branch behavior, correlation, and limits of predictability [6, 12]. Instead of directly adapting the best techniques from branch prediction to block prediction, we undertake a systematic study of how various predictors can effectively predict blocks.

First, we evaluate how scaling and aliasing affect the effectiveness of exit prediction. Then, we study combinations of basic components that may work better compared to working in isolation. We evaluate various types and lengths of histories to understand predictability of exits. For target prediction, we isolate the component-wise mispredictions, suggest simple measures to overcome some of the inefficiencies and characterize the difficult-to-handle mispredictions.

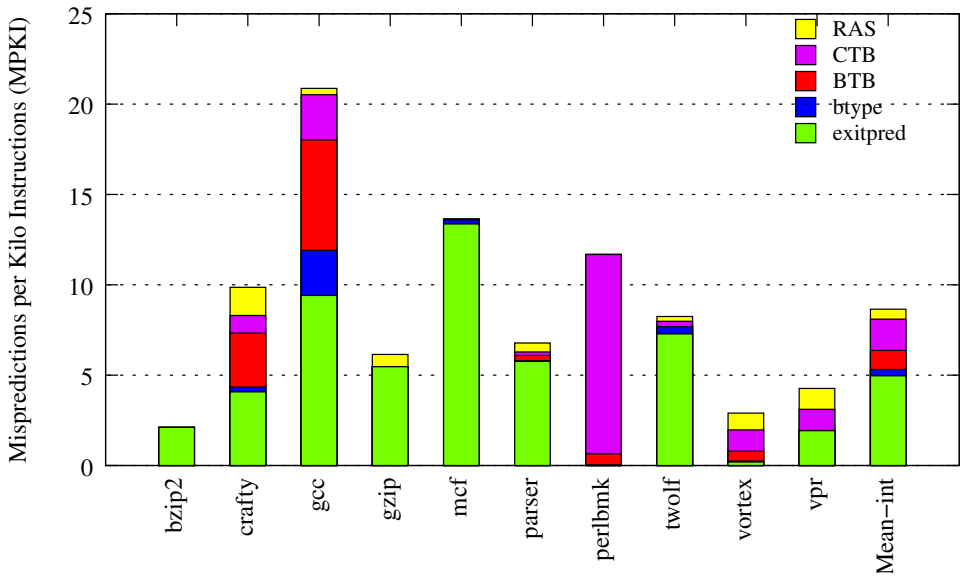
Throughout this chapter, we use the TRIPS functional branch prediction simulator for all our experiments. We use hyperblocks from the Scale compiler (as in Chapter 2) and evaluate different hardware techniques to analyze block predictability. The benchmark

binaries used are the same in all the experiments. In Chapter 5, we analyze the predictability characteristics of hyperblocks and propose solutions to improve predictability.

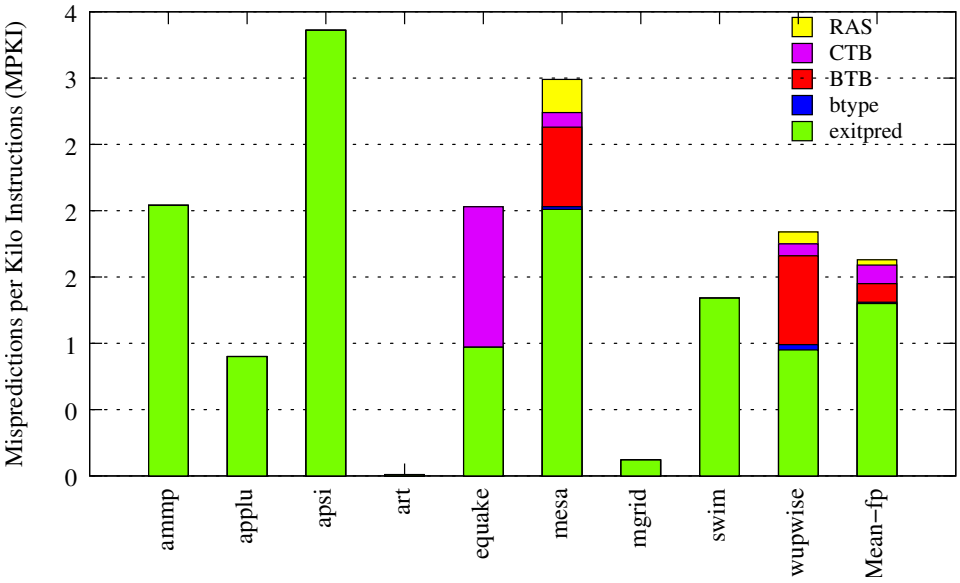
3.1 Misprediction breakdown in the TRIPS prototype predictor

We characterize block mispredictions in the TRIPS prototype predictor by component type. The prototype predictor is divided into five major components to study the misprediction breakdown: exit predictor, Btype predictor, BTB, CTB, and RAS. For this experiment we use the TRIPS prototype 10 KB predictor (5 KB exit + 5 KB target) as well as a scaled-up (approximately) 32 KB predictor (16.8 KB exit + 14.6 KB target). The 32 KB design is a simple scaling of the 10 KB design and contains longer exit histories and more table entries. Along with the 10 KB predictor, we also use the 32 KB predictor for our evaluation as the 10 KB predictor is severely size constrained (in some components) and offers much less scope for the optimization experiments (changes to the predictor design while keeping the size more or less constant). Furthermore, due to technology scaling, future processors will have exponentially higher number of transistors on the die and hence, a branch predictor containing 32 KB of storage (16 KB for exit/branch predictors and 16 KB for target components) is reasonable.

Figure 3.1 shows the component-wise misprediction breakdown (MPKI) for the prototype predictor (10 KB). The lower most component in each bar represents the mispredictions due to the exit predictor. The next component represents the mispredictions from the branch type predictor when the exit predictor has predicted correctly. We only isolate the misses when the exit predictor was correct because the branch type prediction depends on the exit prediction and is almost always wrong when the exit prediction is wrong. Similarly the next three components in the bars represent the three target prediction components (BTB, CTB, and RAS) and their contribution to mispredictions when the exit and the branch type predictions are correct (since target prediction depends on exit and branch type). However, for some benchmarks the total value of the MPKI is slightly greater

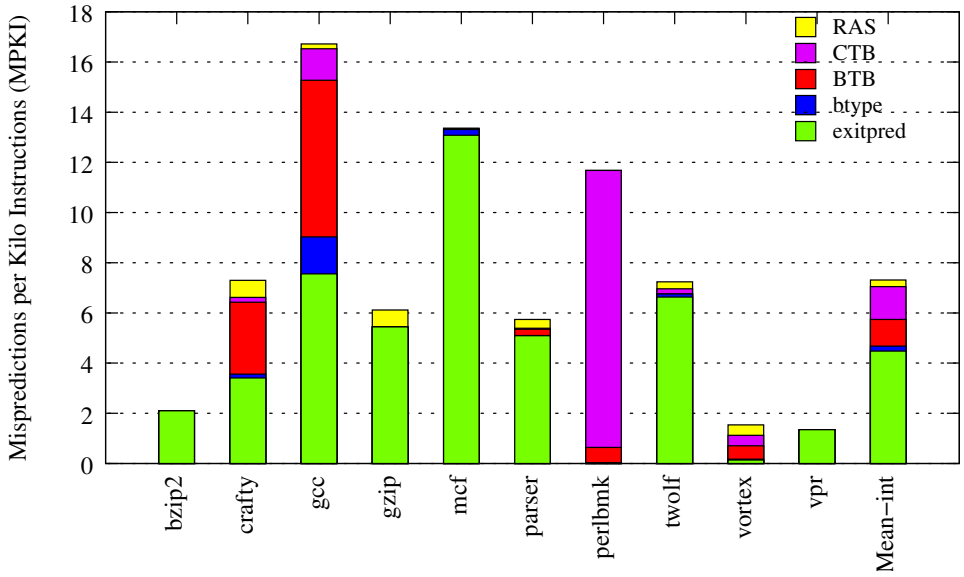


(a) SPEC2K integer benchmarks

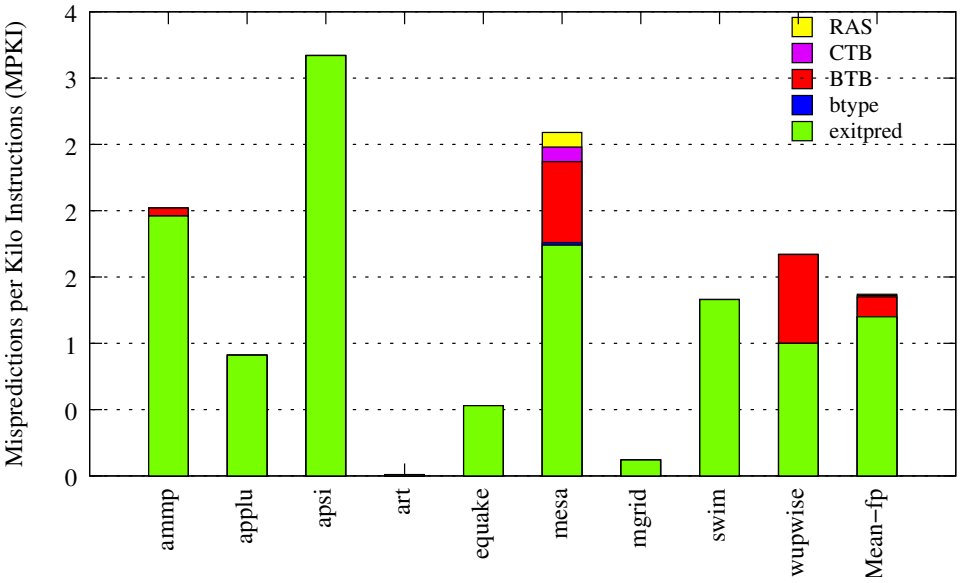


(b) SPEC2K FP benchmarks

Figure 3.1: Prototype predictor (10 KB) MPKI breakdown by component (exit predictor, branch type predictor, BTB, CTB, and RAS)



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 3.2: Prototype predictor (32 KB) MPKI breakdown by component (exit predictor, branch type predictor, BTB, CTB, and RAS)

than the overall MPKI reported in Chapter 2. The reason is that sometimes the target predictor hides the exit predictor inefficiencies; the target is right even when the exit is wrong. However, we isolate the drawbacks in each component and represent the exit MPKI as seen in the exit predictor even though some of the exit mispredictions are hidden by correct target predictions. The graph shows that for most of the benchmarks, the main contributor to MPKI is the exit predictor. The exit MPKI is 4.96 for the integer benchmarks and 1.3 for the FP benchmarks. The major source of exit mispredictions is the due to inefficient prediction technique, sub-optimal predictor design, and poorly predictable hyperblocks. Other factors include aliasing in the prediction tables (contributing to about 18.1% of the MPKI for the prototype predictor and 12.5% of the MPKI for the scaled-up 32 KB prototype predictor) and chooser inefficiency [49]. Using an imperfect chooser contributes to 26.8% of the exit mispredictions for the prototype predictor and 29.7% for the scaled-up 32 KB prototype predictor. For some benchmarks, *gcc* and *perlbmk* in the integer suite and *equake*, *mesa*, and *wupwise* in the FP suite, other components like the BTB and CTB perform poorly.

Figure 3.2 shows the misprediction breakdown for the scaled-up prototype predictor (32 KB). The MPKI of the integer benchmarks improves from 7.76 to 6.49 when increasing the size of the predictor from 10 KB to 32 KB. The exit MPKI is 4.48 for the integer suite and 1.2 for the FP suite. We observe significant reduction in the MPKI for the Btype, CTB, and RAS predictor components compared to the 10 KB predictor. Even in the 32 KB predictor, the significant source of MPKI is the exit predictor. In the following sections, we analyze each of these individual components and determine improvements in predictor design to increase the prediction accuracy.

3.2 Analysis of exit prediction

Exit predictors attempt to predict one of N values (N is eight in TRIPS) while branch predictors attempt to guess one of two values. On average, one access of the exit predictor is equivalent to several accesses of conventional branch predictors. Reducing the number

of accesses has many advantages including longer time available to make predictions, less complexity in the predictor, and reduced power dissipation due to fewer accesses. There are two main disadvantages in exit prediction. First, the "1 out of N" problem is more difficult than the binary prediction problem. Second, there is "less complete" information for the block predictor to make a prediction. For example, the correlation among branches within the current block is not known to the predictor, even though it has correlation information until the last fetched/executed block. Also, hyperblocks hide some difficult-to-predict branches from the predictor using predication and this can lead to misprediction migration [64]. These two reasons can make exit prediction more difficult compared to branch prediction.

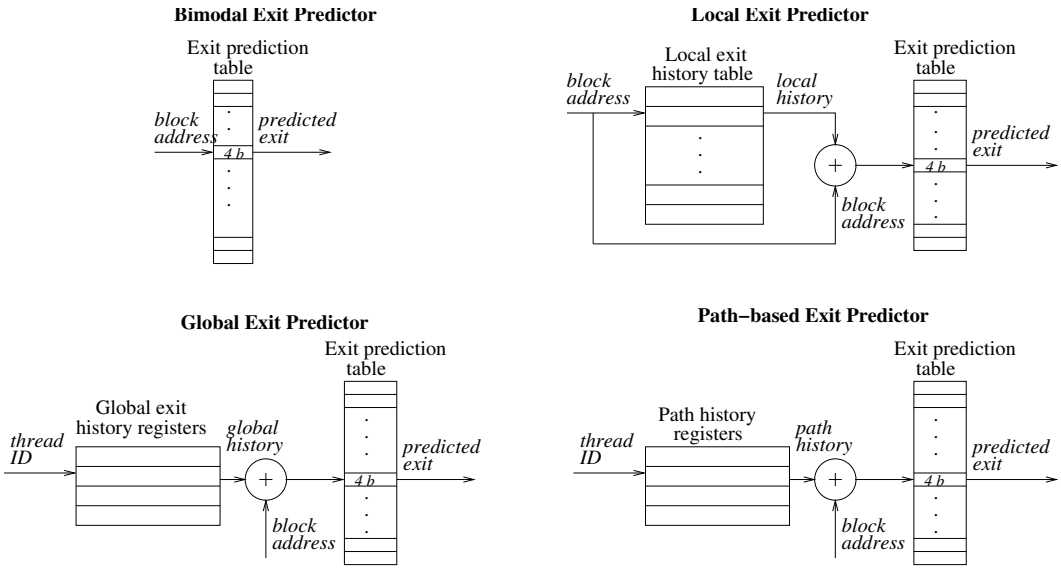


Figure 3.3: Four types of basic exit prediction components: bimodal exit predictor, local exit predictor, global exit predictor, and path-based exit predictor.

Throughout this section, we evaluate four types of exit predictor components, inspired from basic branch predictor component types. The components are shown in Figure 3.3 and described below:

- Bimodal predictor: This exit predictor is similar to the bimodal branch predictor [69]. The prediction table is indexed using the block address bits and the predicted exit is read out of the table. This predictor does not use any history. Typically, the bimodal predictor has been found to give around 80% correct predictions in branch predictors [69].

- Local predictor: This is the local exit predictor as in the TRIPS prototype. The prediction is based on per-block (local) exit histories and the current block address.

- Global predictor: This is the global exit predictor as in the TRIPS prototype. The prediction is based on global exit history and the current block address.

- Path predictor: This is a path-based exit predictor which uses path history (history comprised of address bits from recently encountered blocks). The prediction is based on the path history and the current block address.

The above four predictors are the most commonly found branch direction prediction components. Most of the branch predictors proposed have some combination of one or more of the components above. They may also have more than one prediction component of the same type (with varying history sizes and table sizes). We evaluate these components as individual predictors as well as combinations of predictors. We study the effects of scaling and aliasing on the basic components. Next, we perform a design space exploration of various combinations of predictors to understand which types of predictors are able to predict exits effectively. For these exploration experiments, due to the large number of parameters involved, we perform a reduced design space exploration study by analyzing few parameters, fixing them to the "best" values found and then analyzing other parameters. We describe our assumptions for each experiment in the corresponding subsection.

3.2.1 Effect of predictor size (scaling)

For the first set of experiments, we use the four individual predictors: bimodal, local, global, and path exit predictors. The number of exit bits (for local and global) and address bits (for path) to use in each of histories is determined by evaluating various possibilities for a fixed predictor size (16 KB). These exit bit widths are used in the rest of the predictor sizes explored for the individual components.

Effect of different exit bit widths in history registers

We demonstrate the trade-offs in using different exit bit widths by an example. Figure 3.4 shows the MPKIs achieved by global exit predictors of size 16 KB using a 15-bit global history with various exit bit widths. The figure demonstrates the effect of exit bit width in the history register on predictor MPKI. We show only the SPEC integer benchmarks. The first bar in the figure for each benchmarks shows the MPKI for the predictor using one-bit truncated exit IDs (each exit ID's last one bit is shifted into the global history register) in a 15-bit history. The second bar shows the MPKI achieved when using two-bit truncated exit IDs in the history. The third bar shows the MPKI achieved when using the full three-bit exit IDs in the history. The final bar shows the MPKI achieved by using variable length exits in the global exit history register. For this configuration, the number of bits for the exit to be shifted into the history is calculated as the log (Exit-ID + 1). For example, if the resolved exit ID is 3, we only need two bits to represent this exit in the history ("11") and if the resolved exit ID is 6, we need three bits to represent this exit in the history ("110"). When only one bit is used to represent each exit, the average MPKI of the global predictor is 5.04. Some benchmarks like *gcc* and *mcf* perform very poorly when using a single-bit exit representation. For the two-bit exit configuration, the MPKI achieved is 4.71 and for the three-bit exit configuration, the MPKI achieved is 5.25. Benchmarks like *crafty*, *mcf*, *vortex*, *vpr* achieve the highest MPKI when using three-bit exits. Finally, the variable length exit configuration achieves 4.71 MPKI, same as the two-bit configuration.

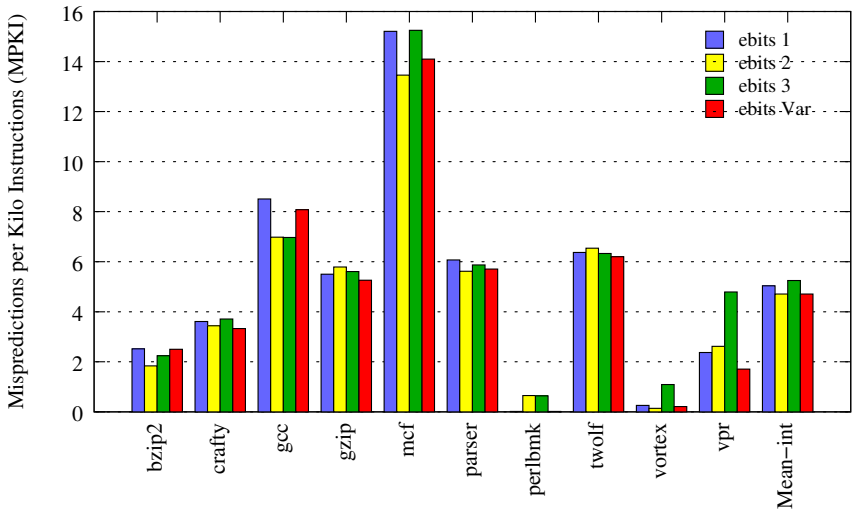
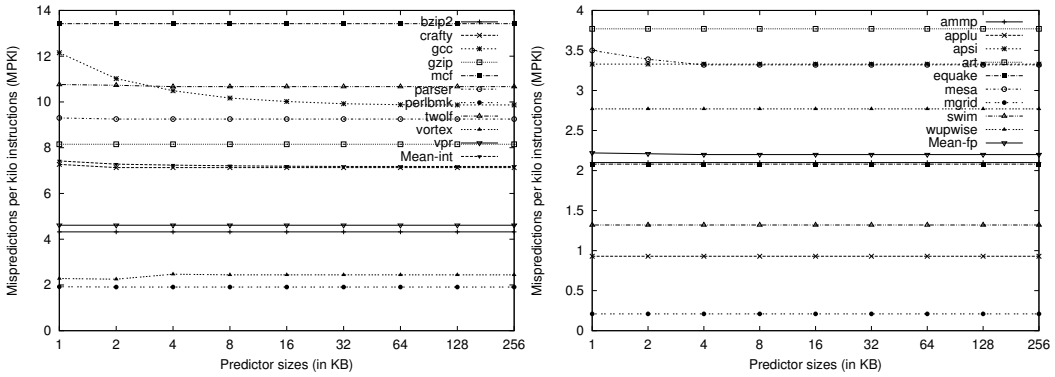


Figure 3.4: MPKI for global exit predictor of size 16 KB using history length of size 15 bits with different encoded exit lengths in history. The first bar corresponds to one-bit truncated exits (15 exits in history), the second bar corresponds to two-bit truncated exits (7.5 exits in history), the third bar corresponds to full three-bit exits (5 exits in history), and the final bar corresponds to variable length exits calculated as log(exit-ID+1) (variable number of exits in history). Results are shown for SPEC integer benchmarks.

The best configurations for the 16 KB global predictor are two-bit exits and variable length exits. Using more bits for representing exits in the global history provides more accurate tracking of exits for correlation but reduces the number of exits that can be stored in the history. Variable length exits are used to remove unnecessary zeroes in the exit history so that more exits can be stored effectively. However, they are slightly more complex to implement than fixed-length exits when considering exit history correction for local predictors and recovery during mispredictions (the number of bits used to represent each taken exit in the history needs to be stored for accurate recovery). Since two-bit fixed-length exits and variable length exits offer the same MPKI, we choose the two-bit fixed length configuration for our global predictors. In all our future experiments, we explore only fixed length exit IDs. Note that these experiments are to demonstrate the trade-offs of using various exit bit widths in the history registers. For the 2nd level prediction tables, we always use the full three-bit exit representation.

Scaling basic exit prediction components from 1 KB to 256 KB

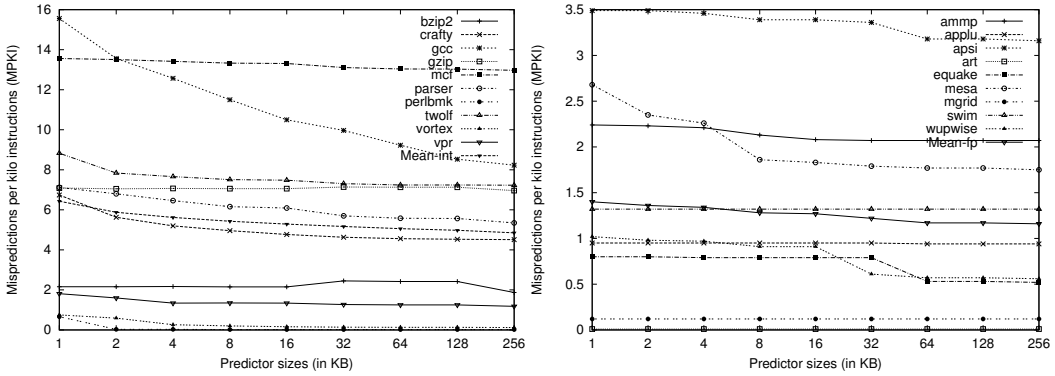
For the scaling experiments, each predictor is scaled from 1 KB to 256 KB. For the local predictor, we choose a reasonable number of L1 table (local history table) entries and scale up only the L2 table (local exit prediction table). A 512-entry L1 table is sufficiently large to avoid most of the aliasing for the evaluated benchmark suite. All predictors are scaled by increasing the number of second-level table entries as well as the history length (N-bit history for a 2^N-entry table). We show exit MPKI results for the four predictors in Figures 3.5, 3.6, 3.7, and 3.9. We also show the mean exit MPKI summary stacks for three predictor sizes (small 2 KB, medium-sized 16 KB, and large 256 KB) in Figure 3.10.



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

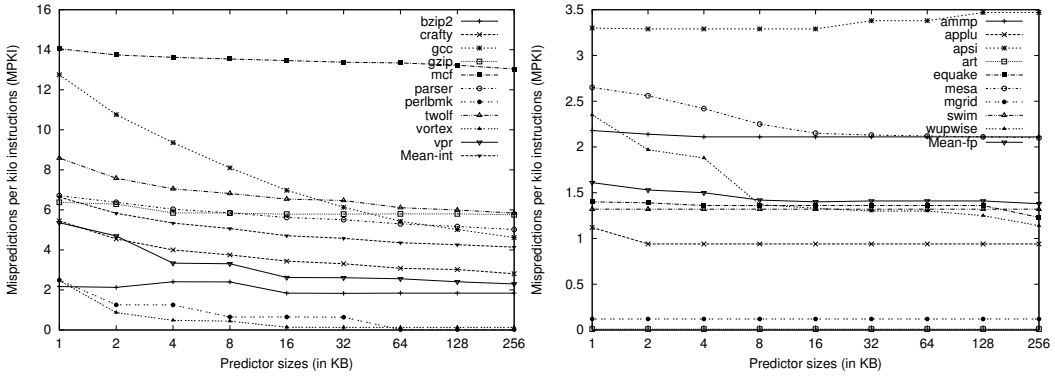
Figure 3.5: MPKI for bimodal (simple address-indexed) exit predictors from 1 KB to 256 KB

From the graphs, we find that for bimodal predictors, there is hardly any benefit in scaling beyond 2 KB except for *gcc* among integer benchmarks and *mesa* among FP benchmarks. The mean bimodal MPKI for the 2 KB size is 7.28 for the integer suite and 2.21 for the FP suite. In contrast, the local predictor benefits from scaling. Though the MPKI reduction is less pronounced beyond 16 KB for integer benchmarks, we see that there is a little improvement due to longer history sizes and reduction of destructive aliasing in the prediction table. Among integer benchmarks *gcc* benefits the most from scaling with an MPKI reduction of almost 50% when scaling from 1 KB to 256 KB. For FP benchmarks



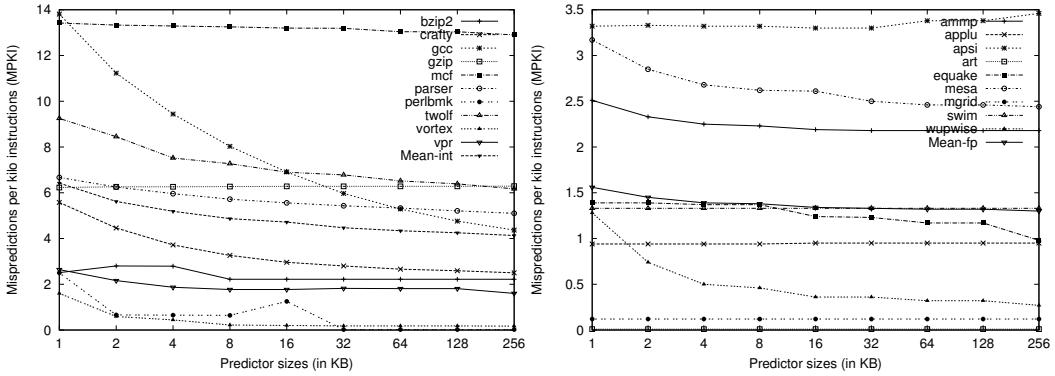
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.6: MPKI for local exit predictors from 1 KB to 256 KB



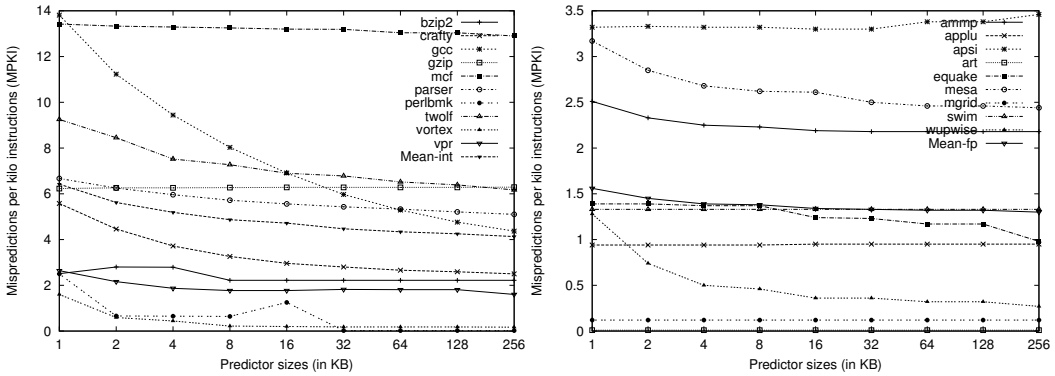
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.7: MPKI for global exit predictors from 1 KB to 256 KB



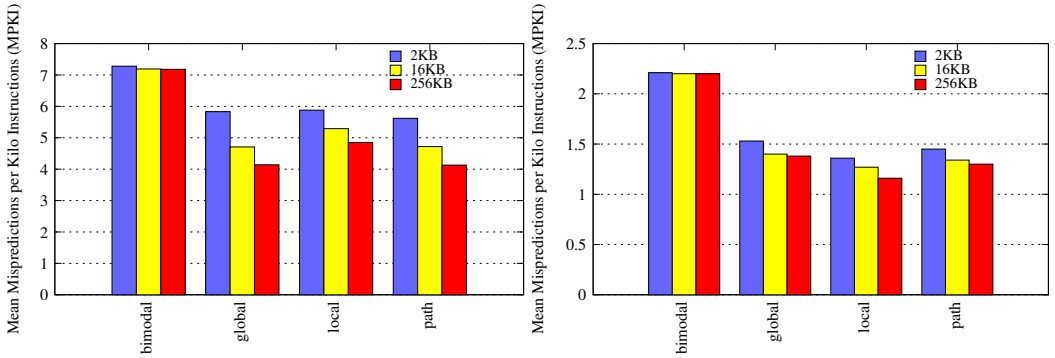
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.8: MPKI for path-based exit predictors from 1 KB to 256 KB



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.9: MPKI for path-based exit predictors from 1 KB to 256 KB



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.10: Mean MPKI for bimodal, local, global, and path-based exit predictors of sizes 2 KB (small predictor), 16 KB (medium-sized predictor), and 256 KB (large predictor).

the mean curve flattens beyond 64 KB. On the whole, there is very little benefit beyond 16 KB for both sets of benchmarks. The 16 KB MPKI values are 5.29 (from 6.43 for 1 KB) and 1.27 (from 1.4 for 1 KB) for the integer and FP benchmarks respectively. The curves for the global and the path exit predictors show a similar trend. Both predictors scale well integer benchmarks. However the curves flatten out beyond 16 KB for FP benchmarks. On average, the path predictor does slightly better than the global predictor. For example, an 8 KB global predictor has an MPKI of 5.34 while an 8 KB path predictor has an MPKI of 5.19 for the integer suite. The difference becomes negligible for predictor sizes 16 KB

and more. The difference in MPKI values between global and path predictors are less pronounced in the FP benchmarks.

The summary stacks in Figure 3.10 show the mean MPKI achieved by each of the four components for 2 KB, 16 KB, and 256 KB configurations. For integer benchmarks, local, global, and path predictors show benefits when scaling up the predictor. When scaling from 16 KB to 256 KB, the local predictor achieves lower benefit compared to the global and path predictor. For the FP benchmarks, the global and path predictors show very little MPKI reduction when scaling to 256 KB from 16 KB. However, the local predictor scales well from 2 KB to 256 KB providing a uniform reduction in the FP MPKI. Overall, the scaling results show that there is significant benefit in scaling global and path predictors, lower benefit in scaling local predictors and hardly any benefit in scaling bimodal predictors. We also find that beyond 2 KB, for the same storage, path and global predictors perform the best followed by local predictor. The bimodal predictor is the least accurate individual component.

3.2.2 Effect of aliasing in prediction tables

Aliasing in the prediction tables (second-level tables in local, global, and path predictors) can be either constructive or destructive. Constructive aliasing results in better prediction for a block than in an interference-free scenario by making entries for a block alias with another entry where the correct prediction for the current block already exists. Destructive aliasing results in poor prediction for at least one of the many blocks aliasing with each other due to their resolved exits being different. We found that this scenario is rare in exit predictors. One possible reason is that a random entry that aliases with another entry might have only a one in eight chance of matching the existing entry whereas with branch prediction it would be 50%. Hence most of the aliasing observed is destructive aliasing.

We analyze the effect of aliasing in the prediction tables by simulating interference-free predictors (simulating one predictor table for each hyperblock). We compare the mean

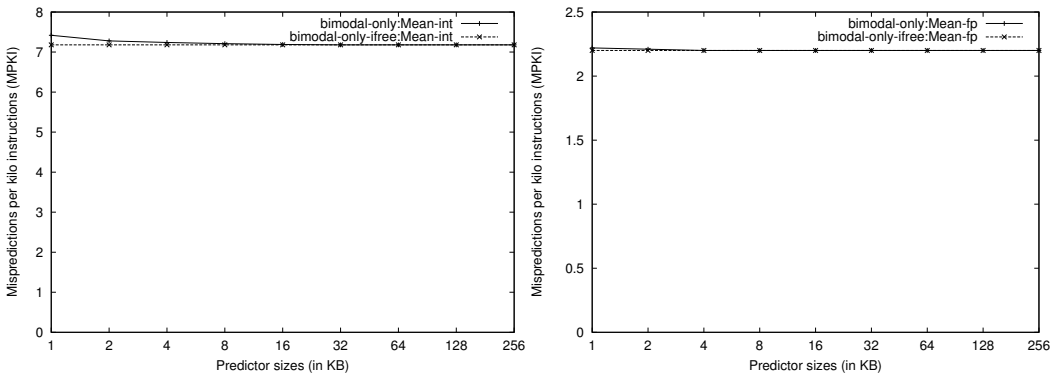
MPKI (separately for integer and FP benchmarks) of each predictor with its interference-free counterpart. Results are shown for nine predictor sizes ranging from 1 KB to 256 KB. Figures 3.11, 3.12, 3.13, and 3.14 show the results of the comparison. The range in the Y-axis is chosen to show the differences in the mean MPKI clearly.

Among bimodal predictors, there is little aliasing in the integer benchmarks (up to 16 KB predictors) and practically no aliasing for the FP benchmarks. Compared to a regular 1 KB bimodal predictor, the 1 KB interference-free bimodal predictor achieves a reduction of 3.2% in MPKI. As the size increases, the relative reduction in MPKI decreases. For the local predictor loss in MPKI due to aliasing is significant for integer benchmarks. Even with a 128 KB predictor, there is 6.8% reduction in MPKI when moving to interference-free predictors. The mean curve corresponding to the interference-free local predictor is almost a straight line whereas the regular local predictor has a curve that shows steady decrease in MPKI with increase in size. This difference shows that the majority of the MPKI reduction when increasing the size of the local predictor (as in the previous set of experiments) comes from removal of destructive aliasing and not from better correlation captured due to longer histories. For the FP benchmarks, the local predictor aliasing is more than 10% for the 1 KB and 2 KB predictors but becomes insignificant beyond 32 KB.

The mean curves for the global and path predictors show a similar trend for integer benchmarks. The aliasing effect is lessened beyond 128 KB in both cases. Compared to the scaling experiments a different trend is observed in the aliasing experiments; the regular path predictor is slightly better than the regular global predictor but the interference-free global predictor is significantly better than the interference-free path predictor. Hence, correlation is better captured by the same size global history compared to the same size path history. However the additional correlation captured in the global history does not make a significant impact on the MPKI as there is more aliasing in the global predictors. There is a likelihood of more aliasing in global exit history depending on the number and types of exits. If several blocks have the first exit in the block taken, then there will be several

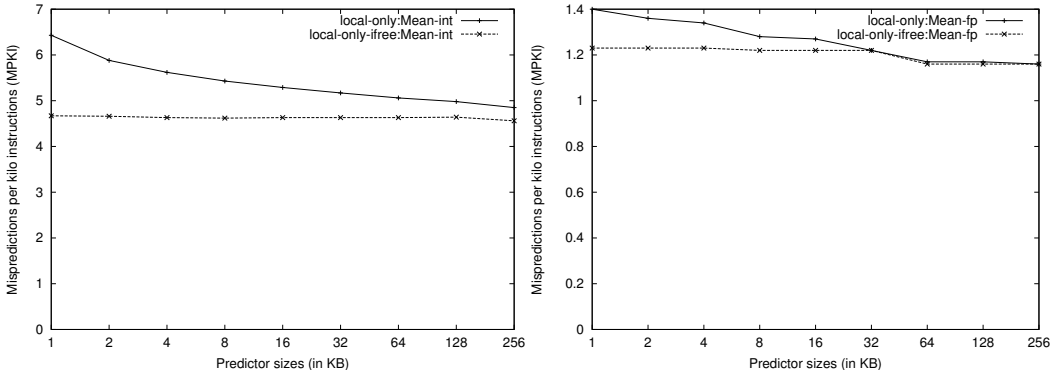
zeroes in the history. Hence the history bits are not uniformly utilized and this can cause more aliasing. For FP benchmarks, the path predictor is marginally better for the regular predictors and gives almost the same MPKI as the global predictor for the interference-free design.

From the interference-free predictor study, we observe that the best individual components are the local and the global predictors at small sizes of 1 KB and 2 KB. For predictors with more state, the global and path predictor components show the lowest MPKI.



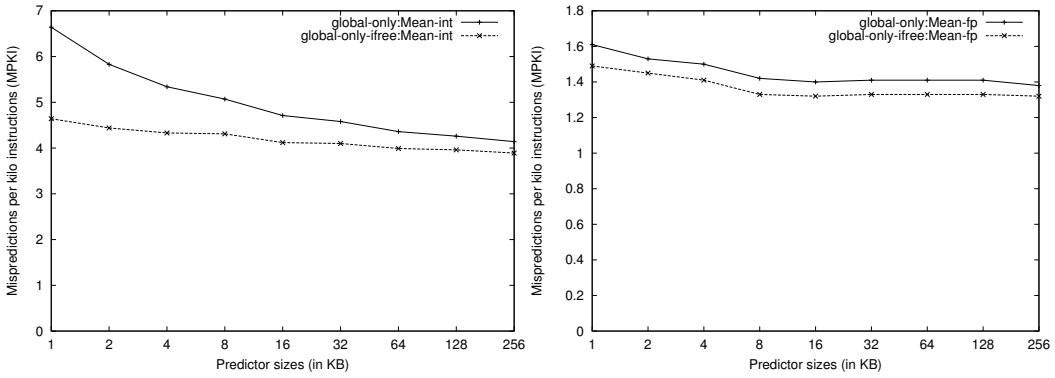
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.11: Effect of aliasing in bimodal exit predictors from 1 KB to 256 KB



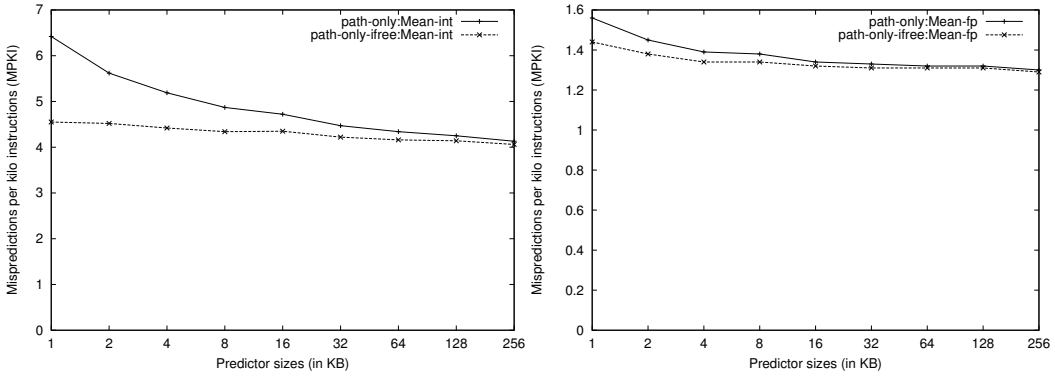
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.12: Effect of aliasing in local exit predictors from 1 KB to 256 KB



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.13: Effect of aliasing in global exit predictors from 1 KB to 256 KB



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.14: Effect of aliasing in path-based exit predictors from 1 KB to 256 KB

3.2.3 Effectiveness of tournament predictors with two components

Tournament predictors [41] use at least two different predictor components along with a chooser to give better predictions than individual predictors of the same size. In this subsection, we consider tournament predictors with two components. The design of such predictors is simple, yet they deliver higher prediction accuracies than individual predictors of the same size. These are the main reasons for choosing the local/global tournament predictor design in the TRIPS prototype processor.

In this subsection, we discuss tournament predictors built from the basic prediction

components presented above. We consider four types of tournament predictors. Bimodal and local predictors are based on self-correlation properties of branches (or blocks) while global and path predictors are based on global inter-branch or inter-block correlation, we choose one self-correlation learning component and one cross-correlation learning component in each of the tournament predictors. Hence the four predictors we simulate are bimodal/global, bimodal/path, local/global, and local/path. These four predictors are shown in Figure 3.15. First, we present the results of scaling for these tournament predictors. Then we determine the best tournament predictor among the four by comparing them directly. Finally we show the effect of choice predictors on tournament predictor performance.

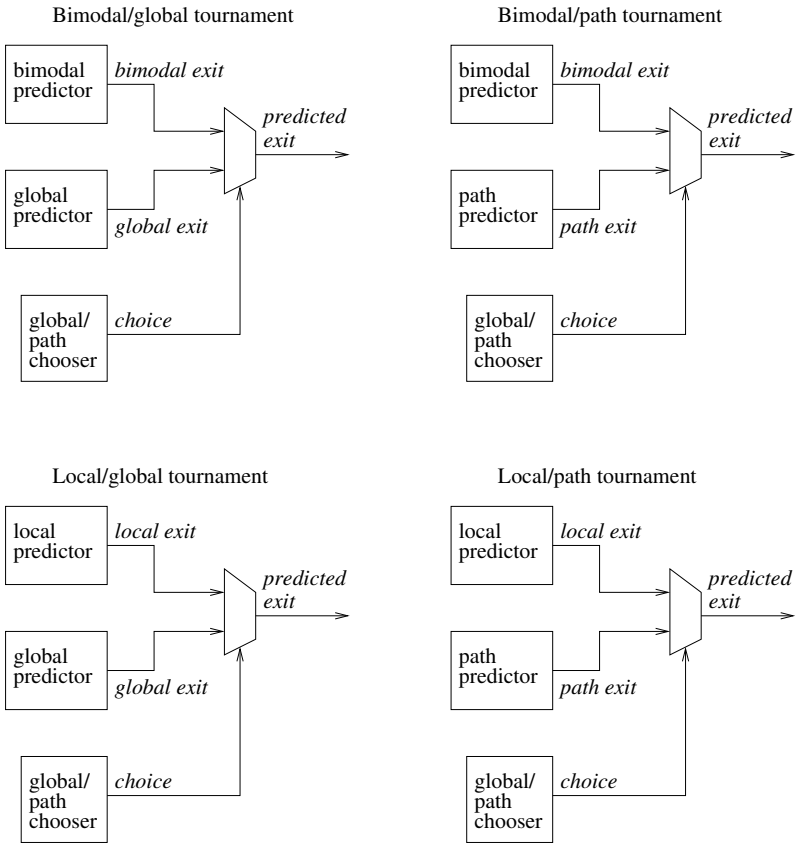
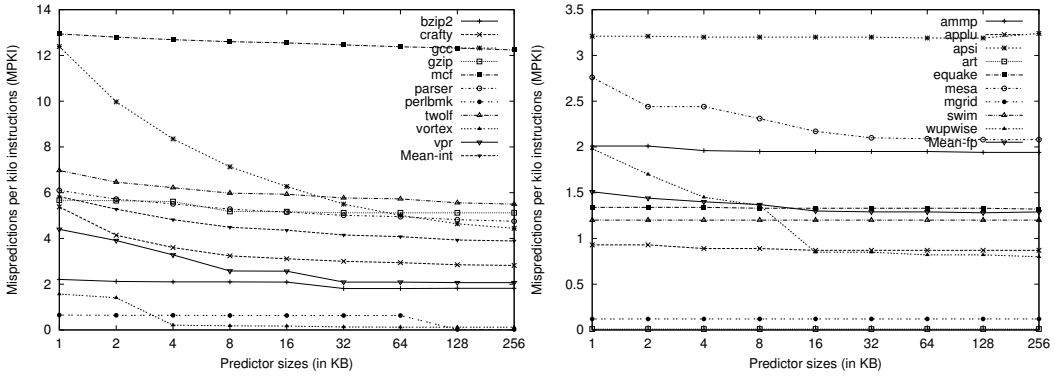


Figure 3.15: Four types of tournament predictors: bimodal/global, bimodal/path, local/global, and local/path.

Effect of tournament predictor scaling

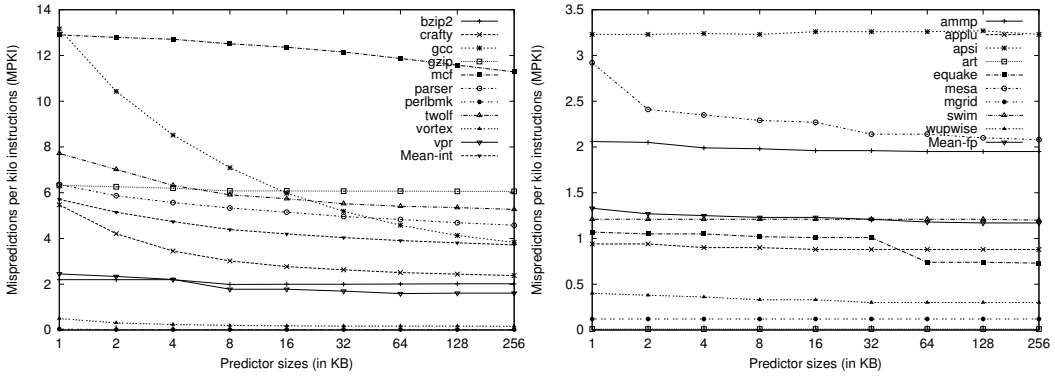
Figures 3.16, 3.17, 3.18, and 3.19 show the results of scaling the bimodal/global, bimodal/path, local/global, and local/path predictors respectively. Exit MPKI results are shown for predictor sizes from 1 KB to 256 KB for the four tournament predictors. In all these predictors, the chooser uses global history to make the choice, as in the TRIPS prototype predictor. The first observation from these graphs is that, all the four graphs for the integer benchmarks are similar to each other. In fact, all the four tournament predictors have almost the same mean MPKI values as the predictors are scaled. For example, for 1 KB predictors, the local/path combination is the worst performing tournament with an MPKI of 6.27. The bimodal/global combination is better with an MPKI of 5.83. The best predictors at 1 KB are local/global and bimodal/path combinations (MPKI of 5.73 and 5.71 respectively). However the differences are less in the 16 KB configuration. local/global and bimodal/path are the best with 4.26 and 4.20 MPKI respectively while local/path and bimodal/global average an MPKI of 4.38 and 4.36 respectively. Difficult to predict benchmarks like *gcc* and *mcf* are not predicted well by any of the tournament predictors when the size is less than 16 KB. On the whole, the local/global and bimodal/path combinations are the best combinations for predicting integer benchmarks.

For FP benchmarks the behavior of some benchmarks varies widely from one combination to the other. For example, *mesa* is predicted well by local/global and local/path combinations. However *applu* is not predicted well by any of the predictors. Beyond 8 KB, the local/global and the local/path tournament predictors are slightly better than the other two. Scaling has little effect for the bimodal/path and local/path predictors beyond 2 KB. In contrast, there is some benefit in scaling for the other two tournament predictors up to 16 KB.



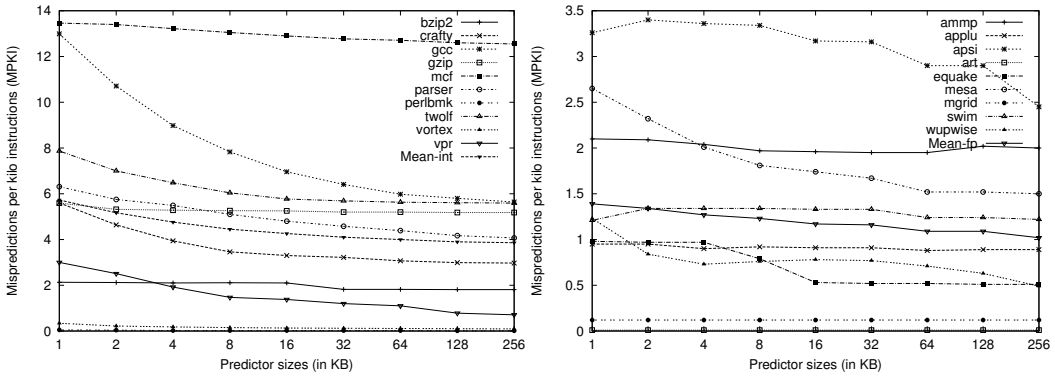
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.16: MPKI for bimodal/global tournament exit predictors from 1 KB to 256 KB



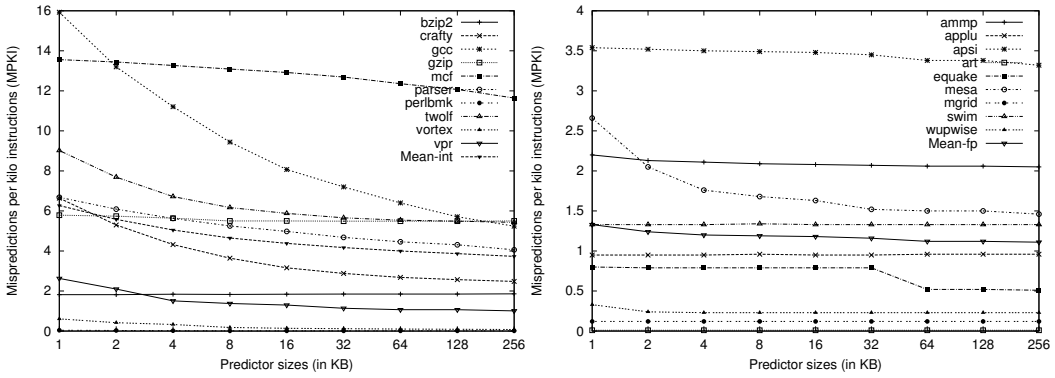
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.17: MPKI for bimodal/path tournament exit predictors from 1 KB to 256 KB



(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.18: MPKI for local/global tournament exit predictors from 1 KB to 256 KB



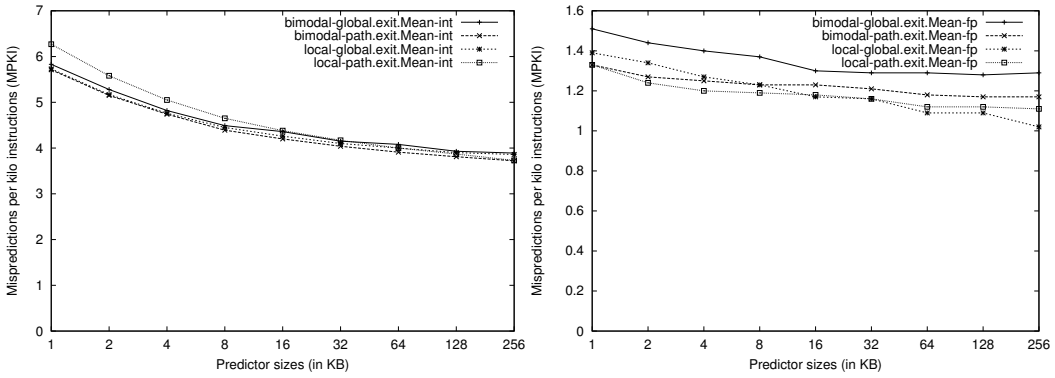
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.19: MPKI for local/path tournament exit predictors from 1 KB to 256 KB

Comparison of tournament predictors

We compare the four tournament predictors discussed above directly by showing them on the same graph. Figure 3.20 shows the mean exit MPKI for the four tournament predictors as the predictors are scaled from 1 KB to 256 KB. For integer benchmarks, the bimodal/path predictor and the local/global predictors have the lowest MPKI, with the bimodal/path combination slightly better than the local/global combination. For FP benchmarks, until 8 KB, the local/path predictor performs the best, while beyond 32 KB, the local/global predictor performs the best. Considering both integer and FP benchmarks, the local/global combination is one of the best tournament predictors. Hence the tournament combination in the TRIPS prototype predictor is one of the best among predictors with similar designs.

To eliminate the effect of chooser predictor mispredictions, we show the same comparison results as above but replace the global-history based chooser with an ideal chooser in Figure 3.21. Notice that that the trends are different. The local/global predictor is the best performing predictor until 16 KB beyond which the local/path predictor is marginally better. For FP benchmarks, beyond 2 KB the local/global tournament is the best.



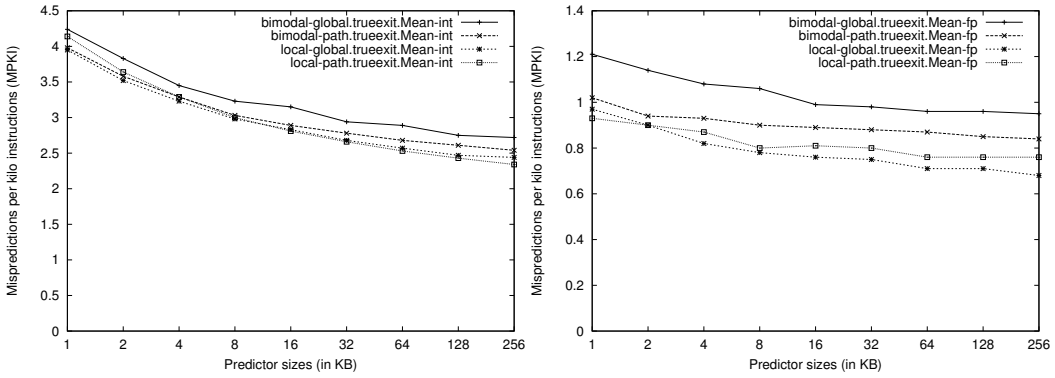
(a) SPEC2K integer benchmarks

(b) SPEC2K FP benchmarks

Figure 3.20: Comparison of exit MPKI using realistic chooser for four tournament predictors from 1 KB to 256 KB

Effect of choice predictors

A tournament predictor's performance depends on the prediction accuracy of the two prediction components as well as effectiveness of the chooser component. Typically a chooser that must select from one of two components uses the block/branch address and (optionally) global history bits to index into a table of saturating counters [4, 41]. The MSB (Most Significant Bit) of the counters indicate whether the first or the second component is to be chosen. We consider the best 16 KB tournament predictors from the previous tournament design space experiments. We evaluate three types of chooser predictors for these tournament predictors: address-indexed, global-history based, and path-history-based. For the global and the path history choosers, we evaluate several exit bit lengths and address bit lengths (respectively) in the histories while keeping the history and table sizes constant. We plot the MPKI from the final best choosers against the default global-history based scaled-up prototype style chooser and an ideal chooser. Figure 3.22 shows the results of the comparison. We see that the MPKI improvement from the prototype chooser to "best" chooser is very small. The graphs also indicate that there is great scope for improving the chooser, as seen in the much lower MPKI values for the ideal chooser. There is a 35% reduction in MPKI in the ideal chooser configuration compared to the "best" chooser for



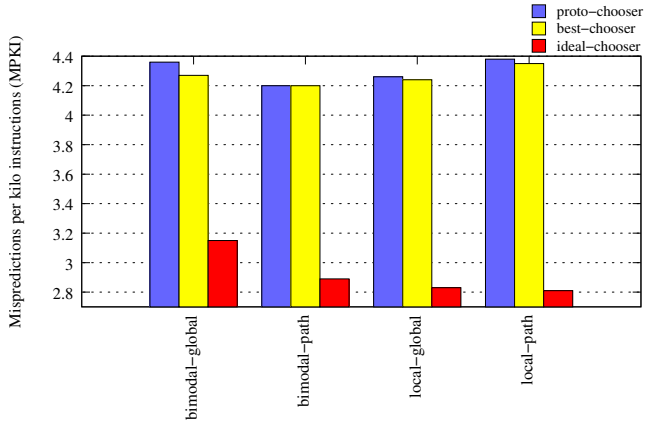
(a) SPEC2K integer benchmarks (b) SPEC2K FP benchmarks

Figure 3.21: Comparison of exit MPKI using ideal chooser for four tournament predictors from 1 KB to 256 KB

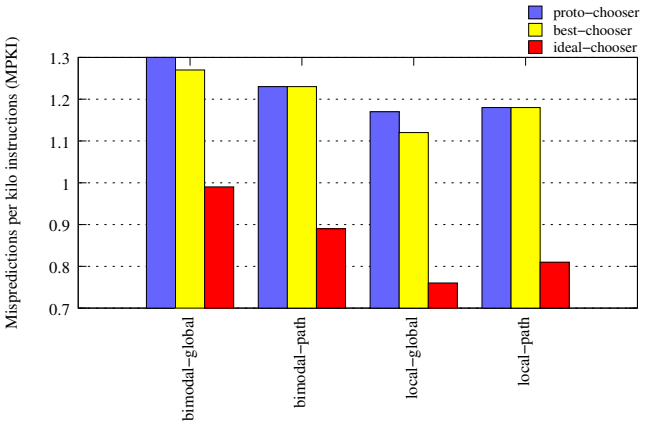
integer benchmarks. For FP benchmarks the difference is about 31%. These results show that better chooser predictor designs can deliver much lower MPKI values even with simple tournament predictors.

3.2.4 Effect of using different types of prediction components

In the experiments from the previous subsection, we found the best tournament predictor of size 16 KB. The best predictors were the local/global tournament and the bimodal/path tournament predictors. However, several other hybrid predictors can be constructed using the same four basic components. In this subsection, we evaluate several such hybrid predictors, all of which have prediction components adding up to 16 KB. We determine the ideal prediction accuracy that can be obtained using such hybrid predictors. Hence, we do not use realistic chooser predictors in this evaluation. In other words, the chooser is *ideal*. The sum of the component predictor sizes excluding the chooser predictor is 16 KB. Our goal is to determine whether the local/global tournament predictor is the best performing hybrid predictor or some other multi-hybrid combination predictor using different types of components exists among predictors of size 16 KB. This experiment does not explore using more than one component of the same type. All the components in each hybrid predictor



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 3.22: Comparison of scaled-prototype (global history) chooser with the best chooser from the chooser analysis experiments and ideal chooser for four tournament predictors

are of different types. Throughout this subsection and the next, we show misprediction rates or prediction accuracies instead of MPKI as it is easier to represent the partitioning of predictions and mispredictions from a total value of 100%. We only evaluate the integer benchmarks for these experiments since they are significantly more difficult to predict and most of them typically require more than one component to achieve good prediction accuracies.

To arrive at the 16 KB hybrid predictor, we use basic bimodal, local, global, and

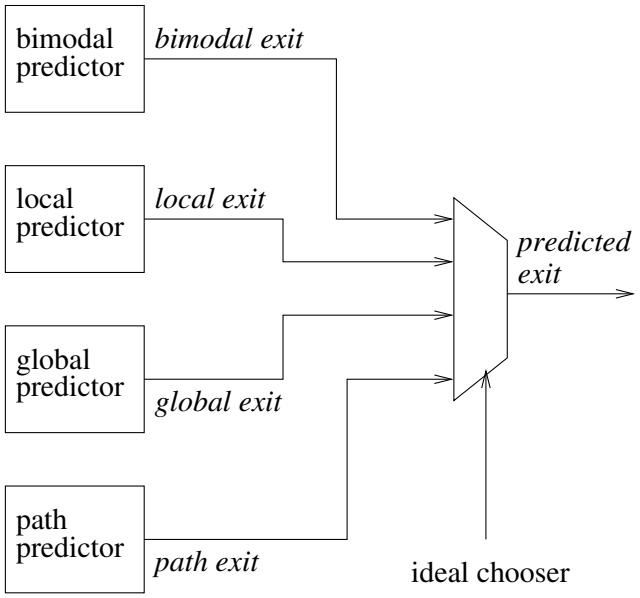


Figure 3.23: Hybrid predictor using four different type of components (bimodal, local, global, and path-based) along with an ideal chooser.

path-based predictor components of sizes 2, 4, and 8 KB. To reduce the design space, we allow only these three sizes for individual components. Figure 3.23 shows the hybrid predictor using an ideal chooser with four types of components (bimodal, local, global, and path-based). The hybrid predictor can have two to four components. Eliminating one or more component types to form hybrid predictors with a smaller number of components is possible. For example, a local predictor of size 8 KB and a path-based predictor of size 8 KB can be combined to form a two-component 16 KB hybrid predictor.

Figure 3.24 shows the mean misprediction rates of all the hybrid predictors that we evaluated. The bars are sorted in ascending order of misprediction rate. Hence the first bar has the lowest misprediction rate, i.e., the highest prediction accuracy. The labels on the X-axis indicate the components chosen along with the size. For example, *B,L,P,G* represents a hybrid predictor with a bimodal component of size *B* KB, a local component of size *L* KB, a path component of size *P* KB, and a global component of size *G* KB.

When a component is not included, its size is indicated by 0. The lowest misprediction rate among this set of 16 KB hybrid predictors is 3.19% for the 2-4-8-2 configuration i.e., hybrid predictor with 2 KB bimodal, 4 KB local, 8 KB path, and 2 KB global predictor components. The highest misprediction rate is 6.31% for the path-global hybrid with each component of size 8 KB. We note that a local/global tournament predictor with equal sizes for the local and global components has a 5.08% misprediction rate, much higher than the best hybrid predictor. Since we do not include chooser inefficiencies here, constructing a choice predictor to choose between two components may be easier than constructing one to choose between four components.

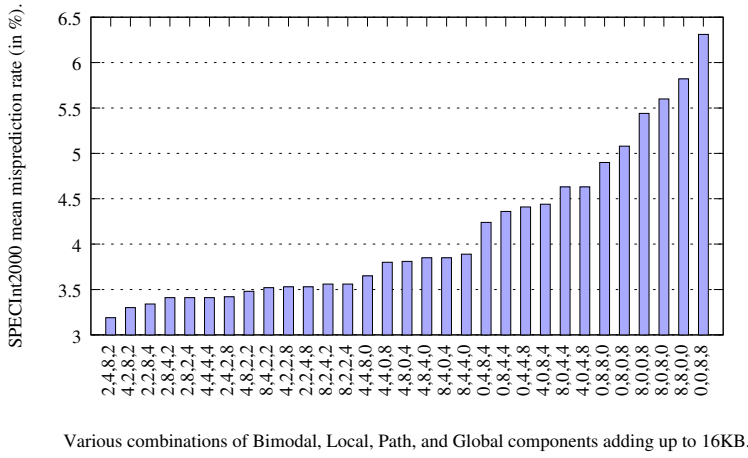


Figure 3.24: Mean misprediction rates for all hybrid predictors (16 KB) using bimodal, local, global, and path components

We use MPKIs to directly compare the best hybrid predictor with the MPKI numbers of tournament predictors. The mean integer MPKI for the best 2-4-8-2 predictor that achieves 3.19% misprediction rate is 1.81. In contrast, the best MPKIs from the previous section for tournament predictors with ideal choosers are 2.83 for local/global tournament and 2.81 for local/path tournament. This comparison is fair since the choosers are ideal in both cases. By moving to a better hybrid predictor of the same size that uses a different mix of components, we achieve a 36% reduction in the MPKI.

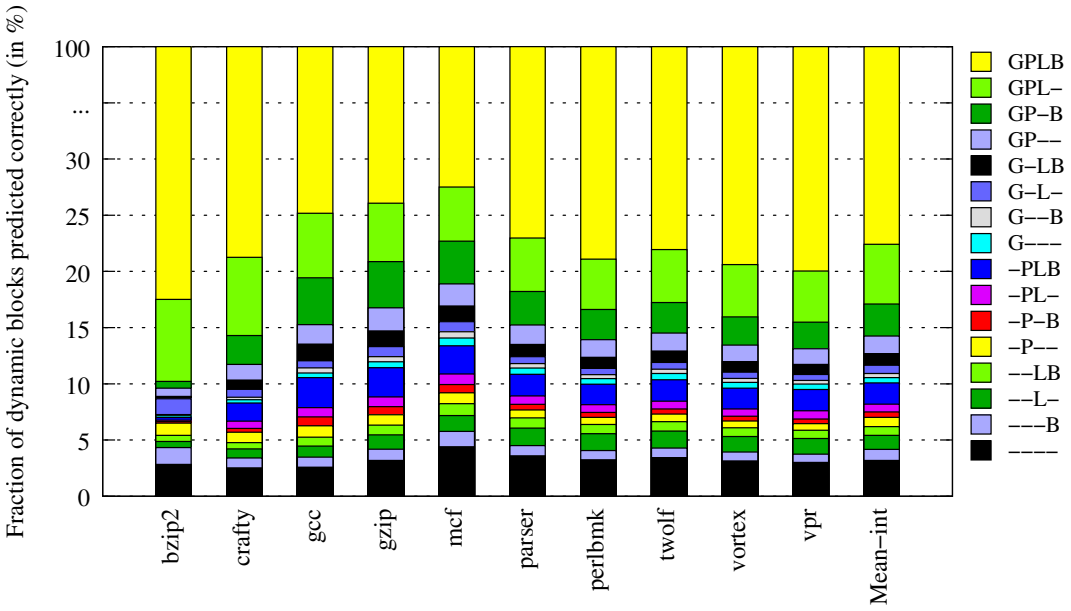


Figure 3.25: Accuracy breakdown in 16 bins for the best hybrid predictor using 2 KB bimodal, 4 KB local, 8 KB path, and 2 KB global predictors.

Figure 3.25 shows the accuracy breakdown for the best 2-4-8-2 hybrid predictor from Figure 3.24. Since we have four components, we have 2^4 combinations of predictors formed with these four components. Each of these combinations is called a *bin*. A bin with ID *i* indicates a particular combination of bimodal-local-path-global (*BLPG*) based on its binary representation. The IDs range from 0 to 15. For example, bin five represents "0101" which indicates the combination of local and global predictors (since the bits corresponding to the positions of local and global are set to one). The value of the bins represents the percentage of predictions that are correct only from the components that the bin's ID represents. For example, bin 0 would represent the percentage of correct predictions that none of the four components can make. In other words, bin 0 stands for the misprediction percentage. Similarly bin 5 stands for the percentage of predictions made correctly only by local and global components. Bin 15 represents the percentage of predictions that are correctly made by all four components. A good hybrid predictor should have combinations

of components that individually may not work that well, but perform much better when combined together. Hence low values for bin 0 are desired. If the individual components are almost as good as the hybrid, there is no reason to choose a hybrid predictor over an individual predictor. The total value of all bins is 100%.

We show the 16 bins for each of the integer benchmarks in Figure 3.25. Since some bins have a low value, the Y-axis has a break to show the bins that sum up to 30%. The rest of the correct predictions come from the 15th bin (i.e., all the components predict correctly). For a majority of the benchmarks, about 80% of the time, all the four predictor components can predict correctly. The bimodal component alone predicts correctly for 1% of dynamic branches. None of the other components are able to make correct predictions for this 1% of branches. By removing the bimodal component from this hybrid, the accuracy will go down by 1%. Similarly, the percentage of branches correctly predicted by local alone is 1.2%, global alone is 0.5%, and path alone is 0.8%. These numbers show the relative importance of each component. We can also draw other inferences from this graph which can be used in constructing hybrid exit predictors made of multiple components. For example, global as well as path can always predict 1.57% of the branches correctly, which local and bimodal cannot. Overall, the graph shows that the potential for achieving higher exit prediction rates is present provided we construct hybrid predictors with the right mix of components and good chooser designs.

3.2.5 Effect of using components with varying history lengths

In the last subsection we evaluated hybrid predictors combining components of different types. We now measure the effectiveness of hybrid predictors with the same type of components. Recent branch studies have looked at using various history lengths to capture various levels of correlation for branches. Typically, in a program, some branches are correlated with branches that occurred before these branches in program order. However some branches may be correlated with branches that were executed much earlier (not recent

branches). For example, correlation between two branches, one present before a loop, and one present after the loop is possible. To capture correlations from different history lengths for different branches, several branch predictors have been proposed [6, 13, 24, 60].

We look at global and path predictors of various history lengths from zero-bit history to 30-bit history. Depending on the number of exits in a block, a 30-bit global exit history can represent more than 100 previously encountered branches. Several branches that would be represented in a traditional global branch history will not be included here because of predication and exit truncation, hiding many branches. This study with nine exit histories of lengths 0 to 30 bits is performed to understand the effect of different history lengths on branch correlation and predictability.

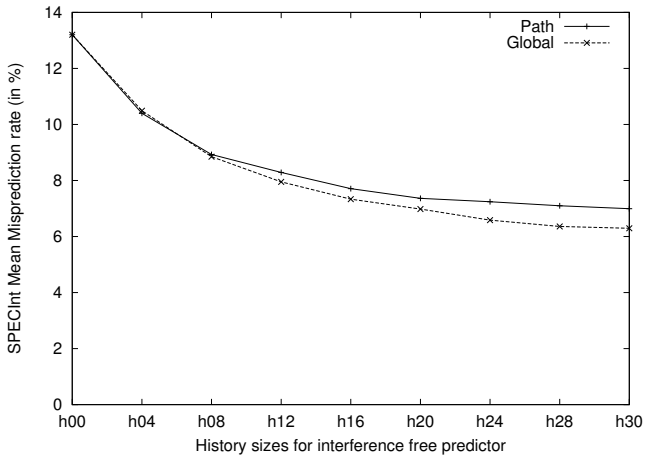


Figure 3.26: Mean misprediction rates for global and path interference-free predictor components using history lengths from 0 to 30

For this hybrid predictor study, we simulate individual interference-free predictor components, a total of 18 predictors, nine for global and nine for path. As in the previous experiment, we use only the integer benchmarks for evaluation. Using interference-free predictors removes the aliasing benefits of large history predictors compared to small history predictors, and retains only the history length benefit for our analysis. We also avoid folding the histories and use them to index the prediction tables as they are stored in the

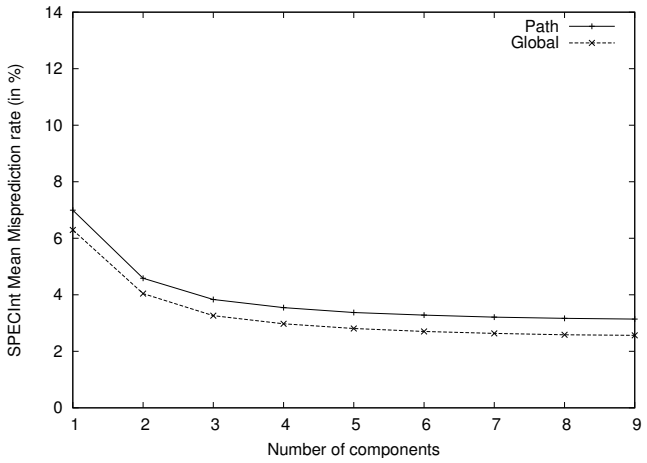


Figure 3.27: Mean misprediction rates for combinations of global interference-free components and combinations of path interference-free components. A total of 9 global predictors (lengths 0 to 30) and 9 path predictors are used. The X-axis denotes the number of components picked out of 9 components. The Y-axis represents the misprediction rate for the best performing configuration in that many number of components. For example, the best four-component global predictor (out of a total of 126 such four-component predictors) gives a misprediction rate of 2.92%.

history registers. In a practical implementation, we may use history folding, smaller tables that can lead to destructive aliasing and fewer component tables to reduce the predictor area. Like the previous subsection evaluating hybrid predictors, we use ideal choosers for this set of experiments also. In Chapter 4 we evaluate designs with realistic choosers.

As in the previous hybrid predictor experiment, we use a similar bin-based methodology to arrive at our results for combinations of predictors. Figure 3.26 shows the mean misprediction rates for each of the predictor components using history lengths from 0 to 30 bits. There are two mean curves, one for global components and one for path components. This graph shows how well each of the individual components perform. The misprediction rate drops from 13.8% for a zero-bit history to 6.4% for 30-bit global history and from 13.8% for a zero-bit history to 7.0% for 30-bit path history.

Figure 3.27 shows the mean misprediction rates for combinations of global predictor components and combinations of path predictor components. Since we use a total of

nine components each in global and path predictors, we can have nine sets of hybrid predictors for each with set *S* corresponding to all hybrid predictors containing *S* components. The figure shows *S* from one to nine, i.e., the number of components in the hybrid predictor. For each set *S*, there can be several possible hybrid predictors. For example, when *S* is two, we can have 36 possibilities for choosing two components out of nine components. Among these 36 hybrid predictors of two components each, we pick the best performing two-component hybrid and plot its misprediction rate in the Y-axis. This is repeated for all nine sets. For both global hybrid predictors and path hybrid predictors, the misprediction rates go down rapidly as more components are included in the hybrid predictor. This effect is purely due to the different history lengths used in the hybrid predictor. When more histories are used, each block has a higher chance of being predicted using the history length that is the "best" for predicting the block effectively. On average, the global predictor performs better than the path-based predictor for all sets.

Num. Components	1	2	3	4	5	6	7	8	9
Best global mispred. rate (%)	6.38	3.97	3.21	2.93	2.77	2.67	2.61	2.58	2.56
Global history lengths chosen	30	0, 30	0, 12, 30	0, 4, 16, 30	0, 4, 12, 20, 30	0, 4, 8, 16, 24, 30	0, 4, 8, 12, 16, 24, 30	0, 4, 8, 12, 16, 20, 24, 30	0, 4, 8, 12, 16, 20, 24, 28, 30
Best path mispred. rate (%)	6.96	4.41	3.68	3.39	3.23	3.14	3.09	3.06	3.05
Path history lengths chosen	30	0, 30	0, 12, 30	0, 4, 16, 30	0, 4, 12, 20, 30	0, 4, 8, 12, 20, 30	0, 4, 8, 12, 16, 24, 30	0, 4, 8, 12, 16, 20, 24, 30	0, 4, 8, 12, 16, 20, 24, 28, 30

Table 3.1: Best global and path multi-component predictor configurations for different numbers of components from one to nine. Misprediction rates are also shown for the corresponding best configuration.

Table 3.1 shows the history sizes of the component predictors for the best performing configuration in each set *S* from Figure 3.27. It also shows the corresponding best configuration misprediction rate. For all sets except when choosing six components, the history lengths included in the best configuration remain the same for global and path exit predictors. The components for each of the best configurations use histories from either end

of the set of histories as well as from the middle of the histories if they can use more. We also observe that more histories are chosen closer to each other at the beginning but when moving towards the longest history, fewer such histories are chosen. This pattern of history selection indicates an approximate geometric trend [60] in the choice of history lengths. The number of branches that require long histories are few, hence fewer components from the longer histories are chosen. Since about 85% of the branches can be predicted well by zero-bit history component itself, the longer history components are useful for few branches. Consider an example from set 5: for a global or path hybrid with five components, three components (0 bit, 4 bit, and 12 bit) are from the lower half of the set of histories and two components (20 bit and 30 bit) are from the upper half.

Figure 3.28 shows the mean misprediction rates of all hybrid predictors that use between three and seven components. The curve for N number of components represents the misprediction rates sorted in ascending order for all the hybrid predictors using N components. There is significant difference between the lowest and highest misprediction rates in all curves. For example, the curve representing the four-component predictors shows that the more than a third of the mispredictions can be reduced when going from the worst-performing configuration to the best-performing configuration. As the number of components in a hybrid increases, the curves tend to be flatter. Thus, when the number of components is higher, even poor combinations of components can perform well. One reason for this behavior is that some components offset the poor performance of other components. In a scenario where we do not have a good benchmark suite representing the applications that will be run on the target microarchitecture, or when extensive design space exploration of the predictor is not possible, we can choose to have more components since they perform well even if the resulting hybrid predictor design does not represent the best optimized configuration.

For the five-component global hybrid predictor, we show the values from the 0th bin which represents the percentage of dynamic branches that none of the components can

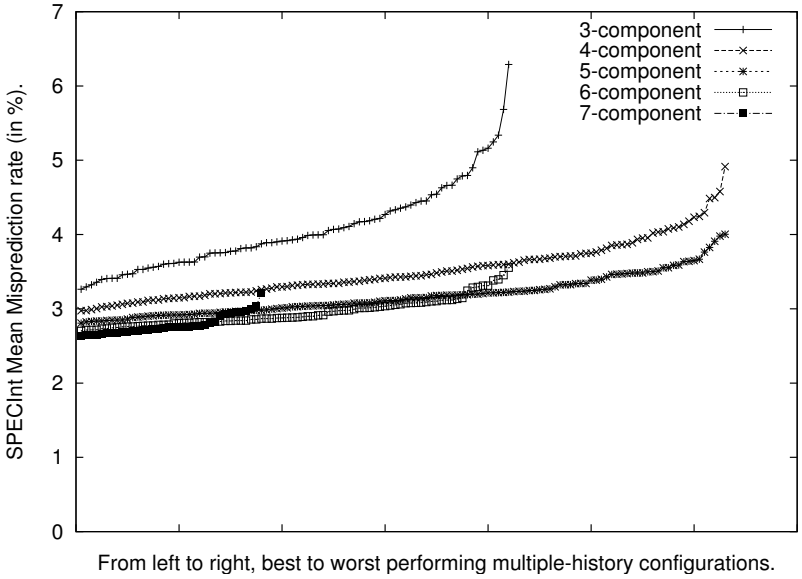
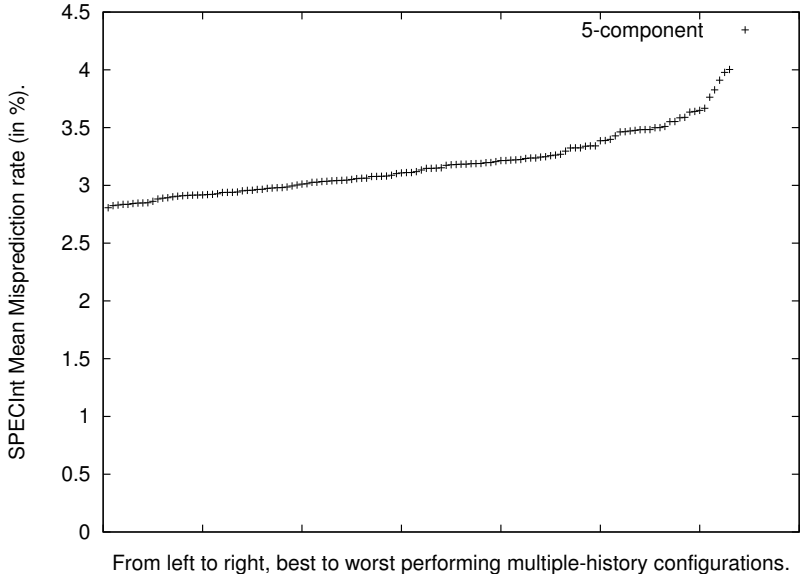


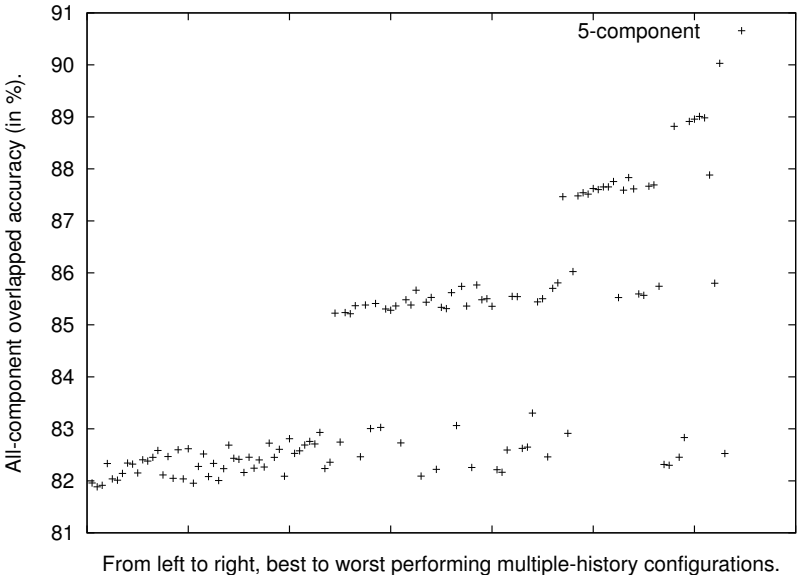
Figure 3.28: Mean misprediction rates for all hybrid predictors using 3 components, 4 components, 5 components, 6 components, and 7 components. The misprediction rates are sorted in ascending order.

predict correctly (i.e., misprediction rates) and the last bin which represents the percentage of dynamic branches predicted correctly by all the five components in the Figure 3.29. The 0th bin curve is the same as from Figure 3.28 but we show the five-component curve alone as an example. The two curves show contrasting trends. The top curve shows the percentage of branches that none can predict correctly while the bottom curve shows the percentage of branches that all can predict correctly. In general, we observe that, as the last bin value increases, the 0th bin value also increases. This effect shows that as the components are more and more similar (all of them can do a majority of the branches), the misprediction rates increase. This trend indicates that choosing components that are less similar to each other can result in hybrid predictors giving higher prediction accuracy.

The best five-component configuration uses histories of lengths 0, 4, 12, 20, and 30. It has a misprediction rate of 2.77% and a mean MPKI of 1.57. The misprediction rate of the best five-component path predictor is 3.23% and the MPKI is 1.82. The MPKI of a



(a) 0th bin: misprediction rate of five-component predictors



(b) Last bin: % of branches correctly predicted by all components

Figure 3.29: Comparison of two bins for all five-component global predictors (chosen from nine history sizes ranging from 0 to 30). The X-axis has the configurations sorted by ascending order of misprediction rate. The top graph shows the misprediction rates (i.e., the rates from the 0th bin which holds the dynamic branches that none of the five components were able to predict correctly). The graph at the bottom shows the accuracy of predictions from the last bin i.e., the dynamic branches that all the five components were able to predict correctly.

five-component path predictor is comparable to the MPKI of a four-component multi-type hybrid predictor from the previous subsection. However the MPKI of a five-component global predictor is even better with a further 13% reduction in the MPKI. For a realistic implementation of these hybrid predictors, the MPKI will be much higher due to aliasing in the predictor tables, history folding, and realistic choosers.

3.3 Analysis of target prediction

Modern branch predictors have sophisticated target prediction components, each tuned to predict one type of branch. Having multiple components can be space efficient: for example, branch targets can be stored as offsets in a target buffer while call targets can be stored as full addresses in another buffer. Further, some branch types need special predictors. Indirect branches are difficult to predict and using a simple BTB to predict indirect branches results in poor accuracy. Similarly, returns are not predicted well by a BTB; a Return Address Stack provides highly accurate predictions for returns.

There are some differences between traditional branch target prediction and the target prediction for TRIPS. First, because TRIPS is a distributed architecture and the predictor does not have access to the branch instructions at prediction time, it needs to predict branch types. Secondly, to support block-atomic call-return semantics, it is necessary to establish call-return relationships at run time. This feature is not required in instruction-atomic architectures. These two aspects were described in the TRIPS prototype target prediction description in the previous chapter.

Our analysis showed that exit prediction is the main source of inefficiency in block predictors. However, for several benchmarks, the target predictor is also inefficient. The target inefficiency is shown in Figure 3.1 for the prototype predictor and in Figure 3.2 for the scaled-up prototype predictor. From these figures, we note that the target predictor contribution to mispredictions is significantly reduced even with simple scaling of the predictor. However, we observe that as the exit predictor improves, more performance bottlenecks in

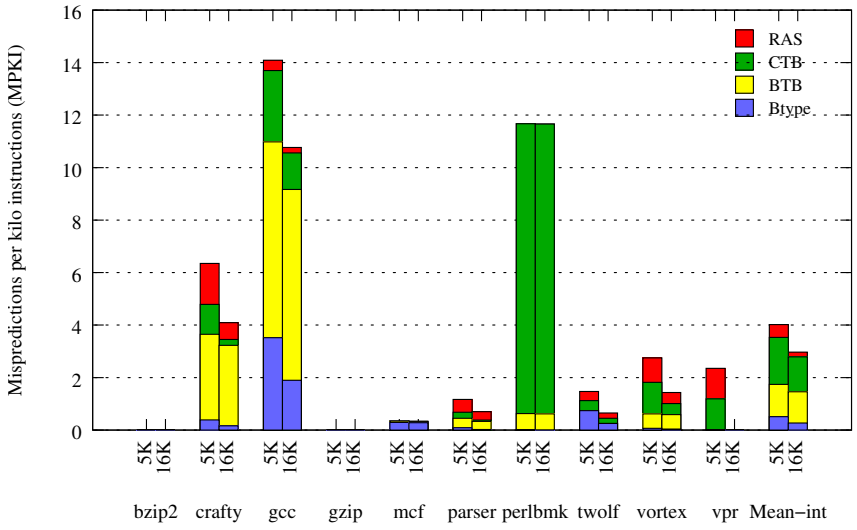
the target predictor may be exposed.

To understand the effectiveness of target prediction, we analyze various sizes and configurations of the multi-component target predictor with a perfect exit predictor. This technique exposes the bottlenecks in the target prediction. As in the exit prediction analysis, we allow a size of 16 KB for the target predictor with a 10% increase or decrease in size depending on the design.

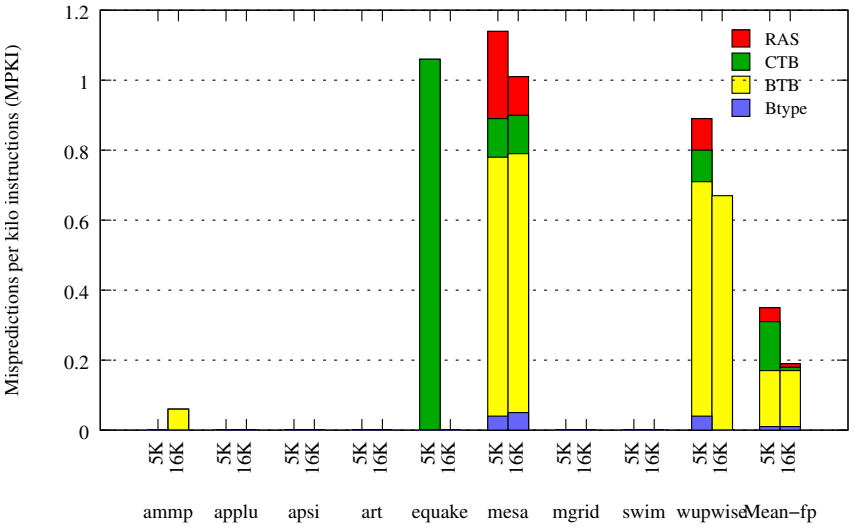
3.3.1 Target MPKI with perfect exit prediction

The MPKI breakdown for the prototype target predictor (5 KB) with a perfect exit predictor and for the scaled-up prototype target predictor (16 KB) with a perfect exit predictor are shown in the Figure 3.30. The first bar for each benchmark shows the MPKI breakdown for the prototype predictor and the second bar shows the breakdown for the 16 KB predictor. Since the exit predictor is perfect, the absolute MPKI values of the Btype, BTB, and CTB components are higher than the numbers seen in Figures 3.1 and 3.2. This is because the realistic exit predictor has the maximum contribution in the overall MPKI and the exit mispredictions hide some of the target mispredictions. When the exit predictor is made perfect, all the problems in the target predictor are now exposed and the target MPKI values are higher.

In the 5 KB predictor, the MPKI values for FP benchmarks is somewhat significant only for the BTB and the CTB. Even these values become insignificant when the predictor is scaled to 16 KB. For the integer benchmarks, the significant sources of mispredictions in most benchmarks are in the BTB and the CTB. The Btype MPKI goes down by half in the 16 KB predictor (0.51 for 5KB and 0.27 for 16 KB); a 4096-entry Btype table was not sufficient to maintain the types of all branches in the current working set. Little reduction in the BTB MPKI is observed when scaling the predictor. However, there is 26% reduction in the CTB MPKI. When scaling the predictor to 32 KB, the number of RAS entries was not changed. But an improvement in the RAS prediction rate is seen in the graph. This



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 3.30: MPKI breakdown for 5 KB (prototype) and 16 KB (scaled-prototype) target predictors with perfect exit prediction for SPEC integer and FP benchmarks. Each stack shows four components (from bottom): Btype MPKI, BTB MPKI, CTB MPKI, and RAS MPKI.

improvement is due to the increase in the CTB size. As described in Chapter 2, the CTB stores the return addresses along with the call target address. The likelihood of the return address being correct is higher in the 32 KB predictor due to the availability of more entries in the CTB to store the return address.

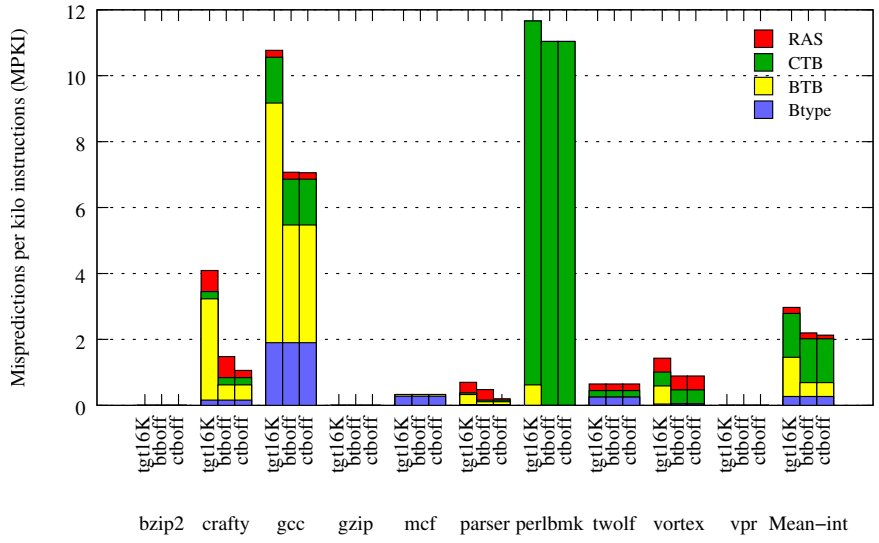
Next, we look at the bottlenecks in each of the components. We only consider the 16 KB predictor for this study. The 5 KB predictor has severe capacity and conflict aliasing and needs more storage to make effective predictions.

Branch type predictor (Btype) MPKI

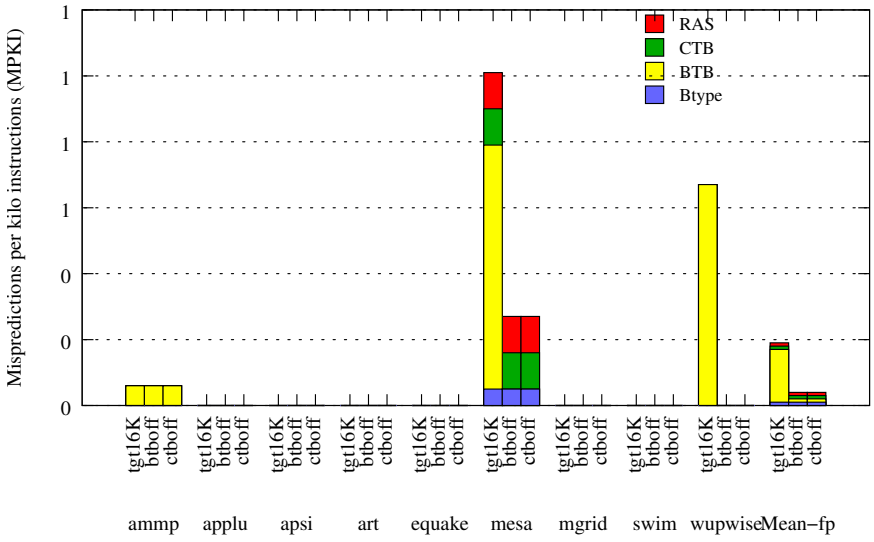
The Btype table is a direct-mapped tagless table. The baseline (16 KB predictor) Btype MPKI is 0.27. Aliasing is present in the Btype predictor but it is found to be low. Our experiments with set-associative Btype tables did not prove to be effective. The main reason for the low effectiveness is the significant tag overhead (at least three bits for tag and three bits for each Btype entry). To keep the size of the Btype predictor constant, the number of entries has to be reduced drastically to accommodate the tag bits. Since the Btype MPKI is not a significant component of the overall MPKI, we propose to maintain the same direct-mapped tagless design as before.

BTB MPKI

Figure 3.1 shows that the number of BTB mispredictions is significantly high, especially for integer benchmarks. The BTB design space exploration showed that offset width and aliasing are the two main problems with the existing BTB. The offset width is important as it indicates the distance from the current block to the next block when a branch is taken. The offset is calculated by finding the difference between the target address and the current block address and shifting right by seven (to remove the last seven zeroes used for block alignment to 128 byte chunks). For example, if the current block address is 0x40000000 and the next block's address if 0x40008800, the signed offset would be 0x110. The offset width of nine



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 3.31: Comparison of MPKI breakdown for 16 KB predictors for SPEC integer and FP benchmarks with perfect exit prediction. The first bar shows the scaled-up prototype target predictor, the second bar shows the MPKI after increasing the BTB offset width, and the third bar shows the MPKI after increasing the return address width in the CTB.

bits used in the prototype predictor and the scaled-up 32 KB predictor was not sufficient to include the targets of several blocks in certain benchmarks. The benchmark affected by the offset width the most was *gcc*. The offset width was calculated based on approximate estimates of the average number of blocks we expect to see in a function. However, if the hyperblock generator in the compiler produces small blocks, the number of blocks to which direct branches may jump (within the same function) will be much higher. Increasing the offset length to 13 bits (from nine bits), removed almost all the mispredictions due to insufficient offset bits.

The results of changing the BTB offset are presented in Figure 3.31. The first bar for each benchmark shows the MPKI breakdown for the baseline 16 KB predictor with perfect exit prediction. The second bar shows the MPKI breakdown after increasing the BTB offset. For integer programs there is 65% reduction in the BTB MPKI. The remaining 35% (MPKI 0.42) can be either due to the absence of a separate indirect branch predictor or due to BTB aliasing. These aspects are discussed in the next subsection.

CTB MPKI

The CTB MPKI for the first bar as shown in Figure 3.31 is 1.33 for integer and 0.01 for FP programs. The CTB stores the full call target address, and not the offset; so offset width mispredictions cannot occur. The CTB MPKI can be caused by aliasing effects and indirect call mispredictions. These aspects are discussed in the next subsection.

RAS MPKI

RAS mispredictions can be caused by an insufficient number of entries in the RAS to track the calls and returns or because of wrong addresses being pushed on to the RAS. Also, since the RAS fix up is not 100% accurate (but very close), the machine may see some wrong RAS fix ups which could lead to mispredictions. In the functional simulator, this problem is not observed due to immediate updates to the predictor. Wrong return addresses can be

pushed on to the RAS if they are not learned in the CTB properly. Hence increasing the CTB offset bit length and reducing aliasing in the CTB can both improve the accuracy of return address prediction. In Figure 3.31, the third bar for each benchmark shows the MPKI improvement after applying both the BTB offset change and the CTB return offset change. The return offset was increased by one bit, which reduces the RAS MPKI from 0.18 to 0.11.

3.3.2 Effect of indirect branches and indirect calls

The two components that contribute the most to MPKI are the BTB and the CTB. Aliasing and poor indirect branch/call prediction are the main reasons for the poor performance of these two components. To isolate the effects of aliasing and indirect branches/calls, we analyzed each exit in every executed block. Not all indirect branches or calls are unpredictable by simple target buffers. Some indirect branches and calls may have only one dynamically taken target. Since this behavior is identical to direct branches and calls, it can be captured easily by the target buffers. Even if there are multiple targets, depending on the pattern in which the targets are observed, the target buffers can give accurate predictions. For example, when the same target is taken several times, a BTB or CTB can easily capture the target information. We track the number of dynamically observed targets for each branch and call exit. If the number of targets for an exit is more than one, the dynamic behavior of this exit is that of an indirect branch/call which may be difficult to predict.

Indirect target distribution

Figure 3.32 shows the distribution of dynamic branches in indirect branches for SPEC integer and FP benchmarks. For each benchmark, we show three stacked bars. The first bar shows the distribution of targets based on dynamic frequency for all indirect control flow instructions. Indirect control flow instructions can be of two types: indirect branches and indirect calls. The second bar for each benchmark shows the distribution for indirect branches along, and the third bar shows the distribution for indirect calls alone. Each

stacked bar shows the dynamic distribution of the targets encountered for indirect branches in the benchmark. The bottommost component in each stack shows the percentage of time the most frequently encountered target is seen for all indirect instructions in the benchmark. The second component from the bottom shows the percentage of time the second most frequently encountered target is seen for indirect instructions. Similarly, the other components show the third, fourth, fifth, and sixth or higher most frequently taken percentages.

For each bar there are two numbers represented as (N, M) on the top of the bar. Here N represents the percentage of all taken exit branches that are indirect instructions of the particular type (all indirect or call or return depending on whether it is the first bar or second bar or third bar respectively). The other number, M represents the misprediction rate of indirect taken exits using the tournament predictor. For several integer benchmarks the value of N is less than 1%. Very few benchmarks like *crafty*, *gcc*, *mcf*, and *perlbmk* have a higher fraction of indirect taken exits. For the FP suite, all the benchmarks have zero or near-zero percentage of indirect taken exits. For this reason, we do not evaluate the FP suite in our indirect predictor experiments in the rest of this chapter and in Chapter 4. Looking at the misprediction rates for the four integer benchmarks that have significant number of indirect taken exits, we conclude that the benchmarks which can significantly benefit from good indirect prediction are *crafty*, *gcc*, and *perlbmk*. For *mcf*, the misprediction rate is much lower compared to these three benchmarks. In *crafty* and *gcc*, indirect branches contribute to almost 100% of all indirect taken exits while in *perlbmk*, indirect calls contribute to almost 100% of all indirect taken exits.

For benchmarks that have a low percentage of indirect branches like *twolf*, *mcf*, and *parser*, the distribution of targets is skewed towards the most common and second most common targets. For the three benchmarks that have a significant percentage of indirect branches, the distribution is not skewed. For *perlbmk*, the distribution is somewhat skewed towards the "rest" category; more than 30% of the dynamic branches take the sixth or higher most commonly taken exit. This also indicates that *perlbmk* has several indirect calls that

have more than five taken targets observed at run time. For *crafty* and *gcc*, more than half of all the indirect branches are found to take the most commonly taken exit. The remaining half of the indirect branches have taken exits that are comprised of the second most common exit, third most common exit and a small fraction of the other categories.

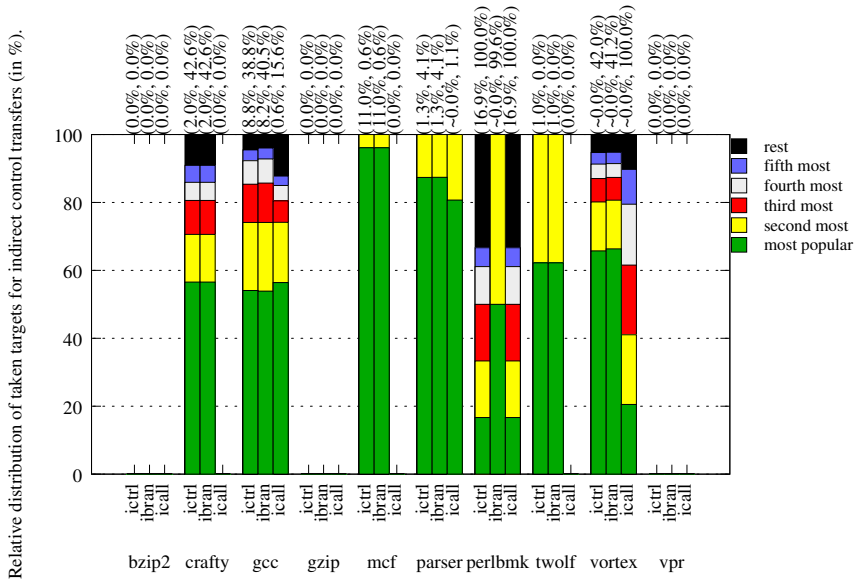
Indirect branch mispredictions

We count the total dynamic mispredictions for dynamically observed multi-target branches. The MPKI is calculated separately for indirect branches and indirect calls (shown in Figure 3.33). The remaining MPKI, other than the indirect MPKI is because of aliasing. For some benchmarks, indirect branches significantly contribute to the MPKI. For example, *gcc* has several multi-target branches that are difficult to predict while *perlbmk* has several multi-target calls that are difficult to predict. These branches need a separate indirect branch predictor to achieve reasonable prediction rates. Only about 25% of mispredictions from the BTB and about 16% of the mispredictions from the CTB can be attributed to aliasing. The effect of aliasing can be reduced by using associative tables instead of direct-mapped tables.

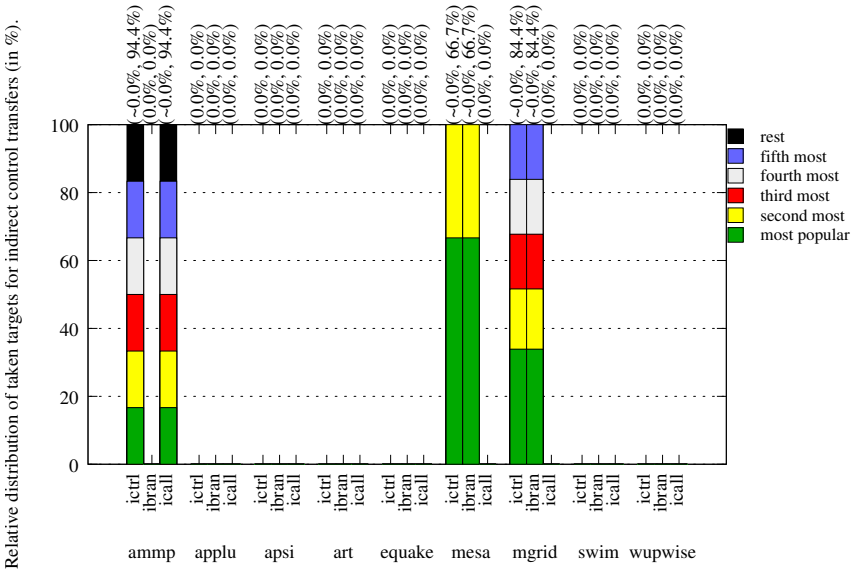
3.4 Summary

In this chapter we highlighted the bottlenecks in the prototype predictor and the scaled-up prototype predictor. We also performed a thorough analysis of various exit predictors to help understand the types of hardware predictors that are effective in predicting exits. For target prediction, we identified the leading cause of target mispredictions in the prototype and scaled-up prototype predictor and motivated the need for a separate indirect branch prediction component in the target predictor. To summarize, the major bottlenecks in the prototype predictor leading to high MPKI are as follows:

- The tournament exit predictor with a simple and easily verifiable design does not



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 3.32: Breakdown of dynamically encountered indirect branches by the number of observed targets for each branch. Breakdown is shown for SPEC integer and FP benchmarks.

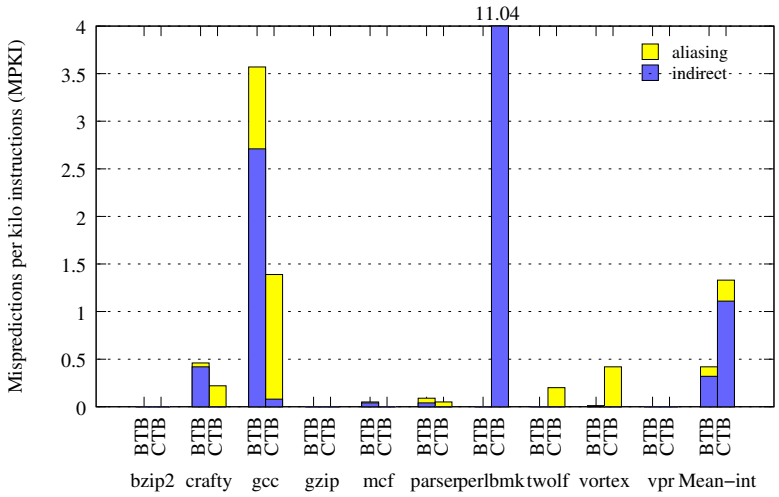


Figure 3.33: BTB and CTB MPKI breakdown for SPEC integer benchmarks with the improved 16 KB target predictor with perfect exit prediction. The first bar shows the BTB MPKI split into MPKI due to indirect branches and MPKI due to BTB aliasing. The second bar shows the CTB MPKI split into MPKI due to indirect calls and MPKI due to CTB aliasing.

perform as well as the current best branch predictors in the literature.

- Small size for the branch type predictor: 4K entries was insufficient to store the types of all the branches.

- Small branch offset length in the BTB (estimated based on Trimaran hyperblocks). After the prototype was implemented, using the tuned Scale compiler, we found that we had underestimated the offset length (calculated based on the approximate number of blocks in large functions).

- Small return offset length in the CTB table.

- Aliasing in the BTB and CTB tables. In general, the CTB entries were sufficient, but some benchmarks with several small functions performed poorly.

- Absence of a separate indirect branch predictor was significant for three benchmarks (*crafty*, *gcc*, and *perlbmk*).

We studied four basic branch prediction components mapped to exit prediction and performed scaling and aliasing analysis for each of individual and two-component tournament predictors. We also analyzed combinations of basic predictor components to see if multi-component exit predictors using different types of prediction components or multiple-component exit predictors using the same type of prediction components (with multiple history lengths). These predictors showed higher potential than the scaled-up 16 KB tournament predictor with ideal design features such as interference-free tables, ideal choosers, and complete histories without folding. Other than exit predictor mispredictions, BTB and CTB mispredictions contribute to the overall MPKI. Increasing the branch and return offset lengths in the BTB and the CTB eliminated a majority of the branch mispredictions and about half of the return address mispredictions. The remaining mispredictions are mainly due to indirect branch and indirect call mispredictions.

Even though the potential for significantly lower MPKI was demonstrated in this chapter using idealized designs, we expect realistic designs to have a much larger MPKI than shown in this chapter. Significant improvements may be possible by intelligent design of the exit prediction components, chooser component, and indirect branch predictor component. In the next chapter, we examine how much of this potential can be tapped by realistic implementations.

Chapter 4

Hardware Techniques to Improve Block Prediction

In the last chapter, we evaluated the bottlenecks in the TRIPS prototype block predictor. We analyzed several types of branch predictor components and explored scaling, aliasing, and predictability of various predictors. The results showed significant potential for improving exit as well as target prediction. In this chapter, we explore hardware techniques to improve exit and target prediction. We present realistic implementations of exit predictors that can perform better than a local/global tournament predictor. First, we design chooser predictors to evaluate the realistic potential of the multi-component hybrid predictor presented in Chapter 3. Second, we present exit predictors inspired from state-of-the-art branch predictors. We discuss the issues in constructing exit predictors similar to branch predictor designs. Next, we present an exit predictor design that can directly use the best binary (branch) predictors. This predictor has the potential to obtain high accuracies using a combination of the best binary predictors and fine-tuned designs for different exit bits. In the previous chapter, we showed that most of the target mispredictions were eliminated by increasing the table size and offset widths. We conclude this chapter by evaluating indirect branch predictors to further improve target prediction.

4.1 Improving hybrid exit predictors using better choosers

Chapter 3 showed the potential of a 16 KB multi-component hybrid predictor using an ideal chooser. We choose the best performing predictor from the previous chapter. The best multi-component exit predictor uses four component predictors of types bimodal (2 KB), local (4 KB), path (8 KB), and global (2 KB). This predictor (called hybrid-4) achieves an MPKI of 1.81 using an ideal chooser (a chooser that always picks a component with the correct exit if any of the components gives the correct prediction). In this section we implement realistic choosers with a maximum size of 3 KB (less than 20% overhead). This chooser has the task of choosing one out of the four component predictors as shown in Figure 4.1.

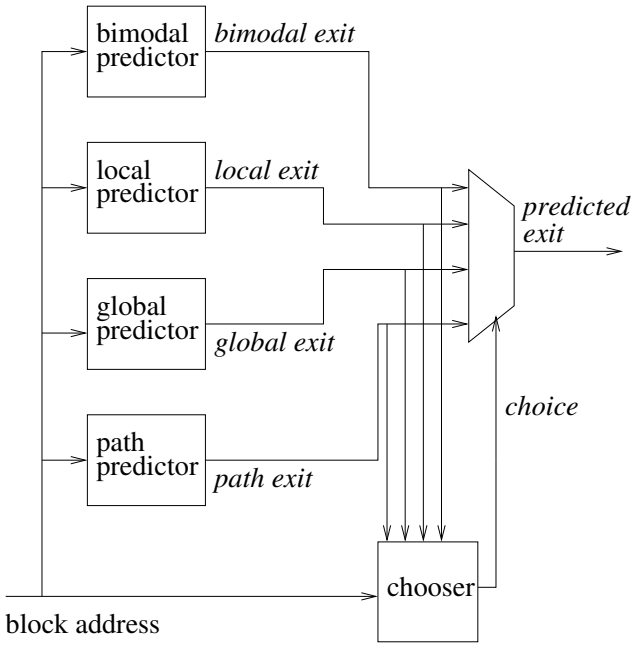


Figure 4.1: Chooser for a four-component exit predictor with bimodal, local, global, and path components. The chooser may use the results of the individual components to make the final choice.

In branch predictors, choosers are one of the most difficult components to implement. The goal of a chooser is to pick the right component every time a prediction is made.

Choosing the right component is a difficult problem because the same component may not always predict accurately for a given block. Depending on the path leading to a block, data inputs to the block, predictor aliasing, and other factors, different components may predict accurately for different instances of the same static block. When there are poorly performing components in a hybrid predictor, the choice becomes more difficult. A good chooser can help achieve lower misprediction rates than any of the individual components. Typically, choosers are built using a combination of branch or block address and global or path history [4, 13, 41]. When there are more than two component predictors, stateless choosers can be employed which use a combination of majority calculation or adders to deliver the final prediction [26, 60]. Our evaluations show that it is more difficult to construct a chooser for a multi-component predictor than designing a chooser for a two-component tournament hybrid predictor. The potential MPKI improvement of our 16 KB hybrid-4 predictor over a 16 KB local/global tournament predictor is more than 35%.

In this section we evaluate various chooser designs without making changes to the component predictors. Several recent multi-component branch predictors use partial updates [62] for the individual components. However, in our chooser analysis, we do not explore partial-update of components to achieve better utilization of the predictor space. The four individual component predictors are always updated for every prediction. We now describe stateless choosers and choosers with approximately 3 KB (or less) of state for the four-component hybrid. We experimented with several choosers for each of the types below. However we show results only for the best choosers in each category. We use the 16 KB local/global tournament predictor using a global-history based chooser as our baseline for evaluation. This baseline predictor with the best global history based chooser (evaluated in Chapter 3 achieves an average integer MPKI of 4.26 and an average FP MPKI of 1.12. An ideal chooser achieves an average integer MPKI of 1.81. The choosers are described below.

Simple hysteresis-based chooser (*C-HYST*)

A simple stateless chooser can be designed by using the hysteresis bits in each of the prediction table entries read out from the four component predictors. The component with the highest value of the hysteresis can be taken as indicating maximum confidence about the prediction. This component can be chosen to deliver the final prediction. However, if there are two or more components with the same highest hysteresis value, a default priority scheme is used to break the tie. In our case, we found that the path predictor performs best followed by the local predictor, the global predictor, and finally, the bimodal predictor. This ordering is used to break ties. The *C-HYST* predictor can perform better when more bits are used for the hysteresis. However, several bits for hysteresis requires significantly more space in the prediction tables.

Majority and weighted majority chooser (*C-MAJ*, *C-WMAJ*)

Another way to build a state-less chooser is to compute the majority of the predictions from the component predictors. The exit ID predicted by the most number of components will be chosen as the majority. This technique is used in several conventional hybrid branch predictors where the sum of the weights have to be greater than a threshold for the branch to be predicted taken [60]. The majority may also be computed including weights (weighted majority). The hysteresis bits can be used as the weights as they indicate the confidence of the prediction approximately. A threshold value is used to determine the final predicted exit. If the highest sum is less than the threshold, a fall-back default component is chosen (in our case, it is the path-based predictor). We evaluated several types of majority and weighted majority choosers with variations in the default component, threshold, and majority weights.

Simple history-based choosers (*C-HISTORY*)

Simple history-based choosers use a history register consisting of any of the basic history types and index into a table entry containing the predicted component ID (two bits for four components) and a hysteresis bit. We experimented with several histories: local, global, path, and various combinations of the above. For some experiments the address bits were concatenated with the history bits while in some cases we XOR-ed the address bits [41]. The best simple history-based chooser based on a concatenation of local, global, and path histories only performed as well as the global-history based chooser.

Fusion-based chooser (*C-FUSION*)

In the previous set of history-based choosers, the hyperblock exit history is used to pick one of the four components. In this set of choosers, the outcome of the individual components are incorporated into the chooser history. One implementation of this technique has been explored by Loh et al. [37] in which they use the outcomes of the individual components to index into the chooser. The chooser is not a traditional chooser which decides the component to be chosen. Instead, it predicts the final outcome using the results of the individual components. We implemented this technique using standalone fusion history as well as fusion and traditional history combinations but found that it was not very efficient for exit prediction. A fusion bit string contains 12 bits, three bits for the predicted exit from each of the four prediction components. We suspect that one of the major reasons for the poor performance of fusion-style hybrid predictors is that the chooser needs to make a three-bit exit prediction instead of a one-bit direction prediction. Further, the predictor needs to be larger than 3 KB to lessen aliasing effects. Because of these effects, we found that a fusion-based chooser is not as efficient as other choosers for small sized choosers. If more storage budget is allocated to the predictor, fusion-based prediction may have higher potential.

Component outcome and history-based choosers (*C-OUTCOME*)

The fusion based predictor described above uses the outcomes from the components to learn patterns of different predictions from the components. These patterns when later repeated, will be found in the fusion predictor table, and the table can make the same prediction as the previously determined outcome. However the fusion predictor is not very efficient for exit prediction. To improve over the history-based and fusion predictors described above, we implemented a history+outcome combination predictor in which we combine few bits from a traditional history like global or path history with a *relative-outcome* bit string from individual components. For four components, we use only six bits to represent the exits instead of the 12 bits used by the fusion predictor. Each of the six bits represents whether every pair of components gives the same predicted exit or not. Since there are six unique pairs among four components, we need only six bits to represent this equivalence. We found that these six bits represent the exit outcomes in a concise way which is useful in determining the final chooser. Using six bits from this outcome history and six bits from global or path history gave higher accuracies than fusion or history-based predictors.

Hybrid majority and history/outcome based choosers (*C-HYBRID*)

In this class of choosers we use a combination of the majority or weighted majority stateless chooser from above with history or fusion or outcome based choosers described above. The majority or weighted majority chooser is used to pick the final component provided the sum of the weights is more than the fixed threshold. For example, consider a simple majority chooser with a threshold of two. Then, if more than two components predict the same exit, it is chosen as the final exit. If not, the outcome or history-based chooser is used as the fall back option. This technique reduces aliasing in the outcome/history-based chooser by making the majority prediction act as a filter to remove easily predictable choices.

Chooser	Description	Int MPKI	FP MPKI
Baseline	global history chooser for local/global tournament	4.24	1.12
C-HYST	hysteresis-based chooser	4.42	1.28
C-WMAJ	weighted-majority with hysteresis weights	4.41	1.37
C-HISTORY	global-history based chooser	4.22	1.19
C-HISTORY	local/global/path-history concatenated chooser	4.21	1.28
C-OUTCOME	6-bit relative outcome with path-history chooser	4.07	1.16
C-HYBRID	weighted majority and fusion+path history based chooser	4.40	1.18
C-HYBRID	weighted majority and outcome-based hybrid chooser	4.06	1.21
C-IDEAL	ideal chooser for four-component hybrid	1.81	0.59

Table 4.1: Mean MPKI for different types of choosers for a four-component hybrid predictor for SPEC integer and FP benchmarks. Baseline is local/global tournament predictor with global history-based chooser. Upper bound is ideal chooser for the four-component hybrid.

4.1.1 Chooser evaluation

We now present the results of evaluation for all the choosers described above. For each class of chooser, we only show the best performing chooser from that class (several choosers were evaluated in our design-space exploration for each class). Table 4.1 shows the results for the predictors described above. The best chooser (when considering integer programs) is the *C-HYBRID* chooser with a weighted majority chooser as the main chooser and outcome based six-bit history (*C-OUTCOME*) chooser (in combination with the path history) as the fallback default chooser. This chooser provides about 4% improvement over the baseline tournament predictor using a global-history based chooser. When considering the upper-bound (as shown in the last row) the results are disappointing. The 4.06 MPKI for integer and 1.21 MPKI for FP benchmarks achieved by the best realistic chooser from our experiments is much lower than the upper bound from the ideal chooser (which has an integer MPKI of 1.81 and an FP MPKI of 0.59). The *C-OUTCOME* chooser does almost as well as the best chooser for integer programs. This indicates that filtering does not have much impact when considering relative outcome-based choosers. The best stateless chooser is the weighted majority chooser with an MPKI of 4.41 for integer benchmarks and 1.37 for FP benchmarks. This represents an increase of 4% in MPKI when compared to the baseline

predictor for integer programs. For FP programs, the best stateless chooser is significantly worse with a 22.3% increase in the MPKI.

Even the realistic hybrid-4 predictor with the best chooser is able to reduce only an extra 4% of the mispredictions when compared to the baseline tournament predictor. There could be several reasons for the poor performance of the choosers including aliasing, insufficient history length, and choosing technique. On the whole, to bridge the gap between a realistic chooser and an ideal chooser, careful study of various choosers and tuning is required.

4.2 Exit predictors inspired from state-of-the-art branch predictors

The TRIPS prototype implementation used a simple tournament exit predictor design with 10-bit local and 12-bit global exit histories. In the last decade, branch prediction research has seen significant improvements by using multi-component long-history based designs. These predictors are highly accurate but have a more complex design than a simple local/global tournament predictor. Since future processors with large instruction windows need highly accurate predictions, it is imperative to improve the performance of control flow predictors by using better techniques.

In this section, we explore exit predictors which are inspired by state-of-the-art branch predictors. We consider three recently proposed branch predictors for our evaluation. The first predictor we consider is Seznec's Optimized Geometric History Length predictor described in [60]. Next, we consider the Tagged Geometric History Length predictor [62]. Finally, we evaluate Jimenez's Piecewise Linear predictor [24]. These three predictors are among the current best branch predictors. All three predictors use intelligent ways of capturing long histories to exploit more correlation than typical global or tournament predictors. We now describe our exit predictors inspired from these three branch

predictors.

4.2.1 OGEHL-inspired exit predictor

While single-history based predictors can give reasonably good accuracies (in the range of 5%-6% misprediction rates for branch prediction), previous work has shown that a combination of history lengths can provide more accurate predictions [6, 60, 71]. In the last chapter, we evaluated combinations of similar-type prediction components (global/path) using several history lengths. We also reported that the best five-component predictor using interference-free global prediction components used history lengths of 0, 4, 12, 20, and 30. It had an ideal MPKI of 1.57 (obtained using a perfect chooser). The history lengths are approximately geometrically increasing with more shorter lengths and few longer lengths. This is the main principle used in the design of a GEHL (Geometric History Length) predictor. We now evaluate an exit predictor design inspired by OGEHL.

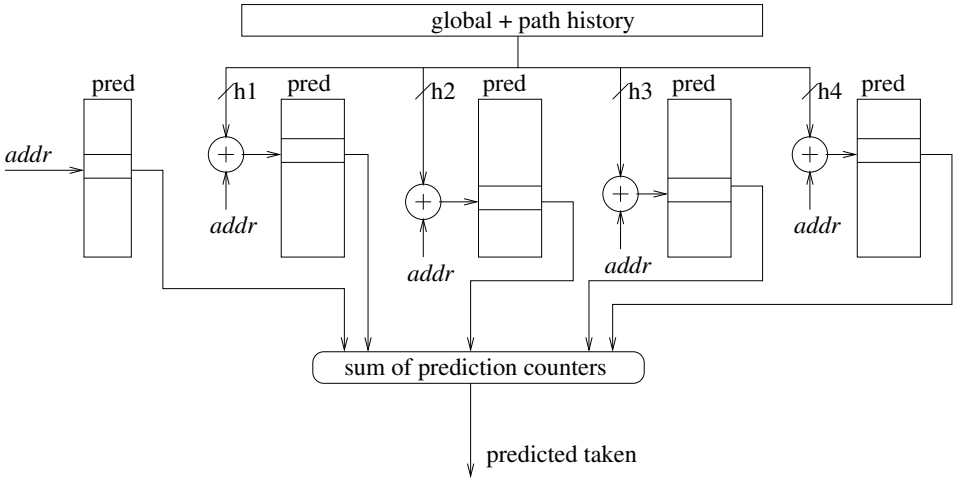


Figure 4.2: A five-component GEHL branch predictor that consists of a bimodal component and four other global+path history indexed components with geometrically increasing history lengths (*h1*, *h2*, *h3*, *h4*). The final direction prediction is made by weighted majority sum of the individual saturating counters. If the sum is more than the threshold the predictor predicts taken.

The GEHL branch predictor [60] uses a set of geometrically increasing history

lengths to index several prediction tables. For example, a six-component GEHL predictor can have history lengths of 0, 3, 9, 27, 81, and 243. A five-component GEHL branch predictor is shown in Figure 4.2. The first component table is usually indexed without any history as in the bimodal predictor. Typically, the index for a component is constructed using a hash of the global, path, and branch address bits. Typically, to generate an *N-bit* index into the prediction table, *N* bits each are picked from the global, path, and address bits. The global histories are of various lengths for various component tables. Hence, to obtain *N* bits from a history with length more than *N*, some bits are selected from the history at regular intervals (skipping some bits regularly). The three *N*-bit values (address, path, global) are hashed together to produce the final index. Each of the second-level prediction table entries contains a saturating counter. The final branch prediction is computed using an adder tree by adding the values of all the counters. If the sum is greater than a threshold, the branch is predicted taken, else it is predicted not taken. The OGEHL predictor optimizes the GEHL design further by using dynamic history length fitting (to enable history lengths to change based on program behavior) and dynamic replacement threshold fitting for replacement of component table entries. To simplify our exit predictor design, we only used the GEHL predictor and did not implement the optimizations present in OGEHL. Seznec [60] reports that using dynamic history length fitting results in an average improvement of 6-7% over fixed history lengths while dynamic threshold fitting provides about 2-3% improvement over a fixed threshold.

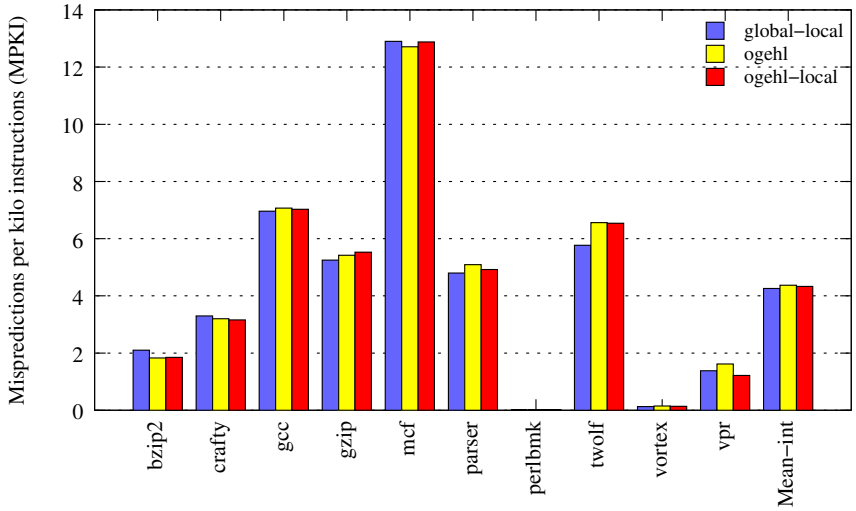
The mapping of the GEHL branch predictor to exit prediction is not very straightforward. The main issue is that most modern branch predictors including the GEHL predictor eliminate the chooser predictor and use an adder-function to determine the branch direction. Since direction prediction is a binary function, summation of values is meaningful. For block prediction, we had to re-design this component. We use majority vote counting with a threshold *N* and a backup chooser. If more than *N* components predict the same exit, we use that prediction as the final exit prediction. If not, we use a chooser which chooses

one of the component predictors. This chooser stores the component ID and hysteresis bits. It can be indexed like any of the history-based choosers we described in the previous section. Another difference between the GEHL predictor and our GEHL-like exit predictor is in the replacement of component table entries. We evaluated both partial updates (by selecting only some tables for updates based on thresholds and predictions made) and complete updates. We found that for the GEHL-like exit prediction, complete updates work better than partial updates.

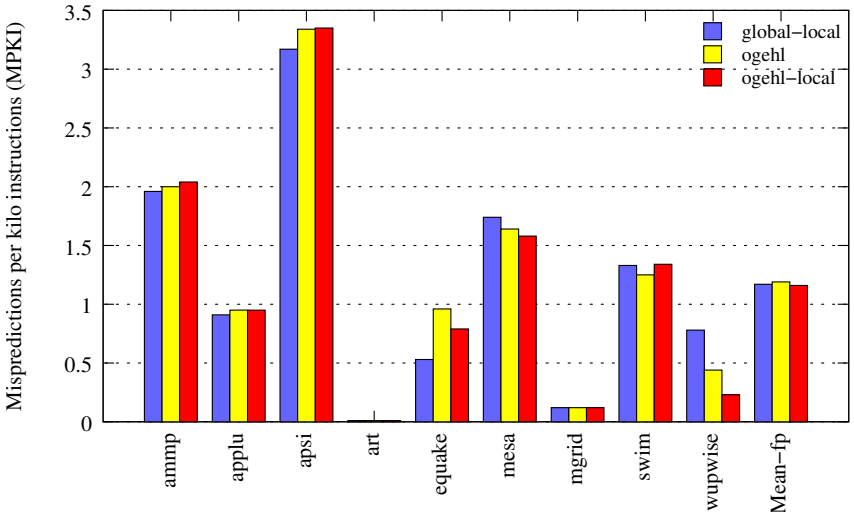
The other difference between the GEHL branch predictor and our mapping is in the component predictor types. The GEHL branch predictor is a global-component-only predictor. It may include a bimodal component but does not include a local predictor component. For hyperblocks though, we found that local exit predictors are very important (in some cases even a global predictor with twice the size of a local/global hybrid fails to deliver as much accuracy) due to the presence of some global correlation within the blocks (also discussed in Chapter 5). Secondly, as stated in the previous chapters, implementing local predictors for hyperblocks is much less complex. Hence, we include a local component and use a separate regular global-history based tournament chooser to choose between the local and the GEHL exit predictors.

GEHL exit predictor evaluation

We experimented with several table sizes, history lengths, threshold values, choosers, and with /without local components. We also considered different numbers of components. We summarize the best results from our evaluation in Figure 4.3. As stated above, we use the local/global tournament exit predictor as our baseline for comparison. In Figure 4.3, for each benchmark we show the MPKI of the local/global tournament predictor, the MPKI achieved by our best five-component GEHL-like predictor and the MPKI achieved by a tournament predictor with local and best five-component GEHL-like predictor components. All predictors are approximately 16 KB in size. The results are disappointing. The GEHL-like



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.3: MPKI for local/global tournament predictor, GEHL-like exit predictor, and local/GEHL tournament exit predictor of size 16KB for integer and FP benchmarks.

predictor and the local/GEHL predictor have marginally higher MPKI values than the local/global tournament predictor for integer benchmarks. The local/global tournament has an exit MPKI of 4.26 while the GEHL-like predictor has an MPKI of 4.37 and the local/GEHL tournament has an MPKI of 4.33. On the other hand, for FP benchmarks all three perform almost on par with each other. The local/global tournament predictor has an MPKI of 1.17 while the GEHL-like predictor has an MPKI of 1.19 and the local/GEHL predictor has an MPKI of 1.16.

We investigated the reasons for the poor performance of the GEHL-like exit predictor. The GEHL predictor uses longer histories and the local/GEHL predictor uses local correlation information also along with global correlation. Still, the GEHL-like exit predictor has a major drawback: it does not rely on majority choice alone to make a prediction like the corresponding branch predictor. It uses a separate chooser predictor and the inefficiency of the chooser predictor causes the majority of the mispredictions.

4.2.2 TAGE-inspired exit predictor

The TAGE predictor [62] uses a set of geometrically increasing global histories similar to the GEHL predictor to access various tagged tables. It is inspired from the ideal PPM predictor [6] that uses all history lengths from 0 to N for an $N+1$-component prediction consisting of Markov predictors. The maximum history length used is N. The GEHL predictor, on the other hand, uses geometrically increasing history lengths (*O(log N)* histories indexing *O(log N)* tables, for a maximum history length of N). By combining the geometric history length approach of GEHL with the ideal PPM approach, the number of components and histories is drastically reduced and hence, an implementable version of the PPM predictor, i.e., TAGE is obtained. The prediction tables are tagged to help ensure that a component giving a good prediction for a branch is the only component making a prediction for that branch. In a practical design, often, two components are updated. Hence prediction for a branch occurring in a particular path is never made by more than two tables. This design

saves space in the other components for other branches and reduces replication of predictions (as in OGEHL, for example). However, tags occupy lot of space and a fine balance is required between the number of table entries and tag length. When a branch can be predicted well by a component using a short history length, it is not allocated to longer-history components. This optimization reduces destructive aliasing and helps the longer-history components predict the difficult branches better. Typically, the first component is a tag-less bimodal predictor. When a branch needs to be predicted, the component for which the tag matches gives the final prediction. If more than one component has a matching tag, the longer-history component is chosen. When none of the components have matching tags, the bimodal component makes the prediction. TAGE can also use dynamically varying histories and replacement thresholds.

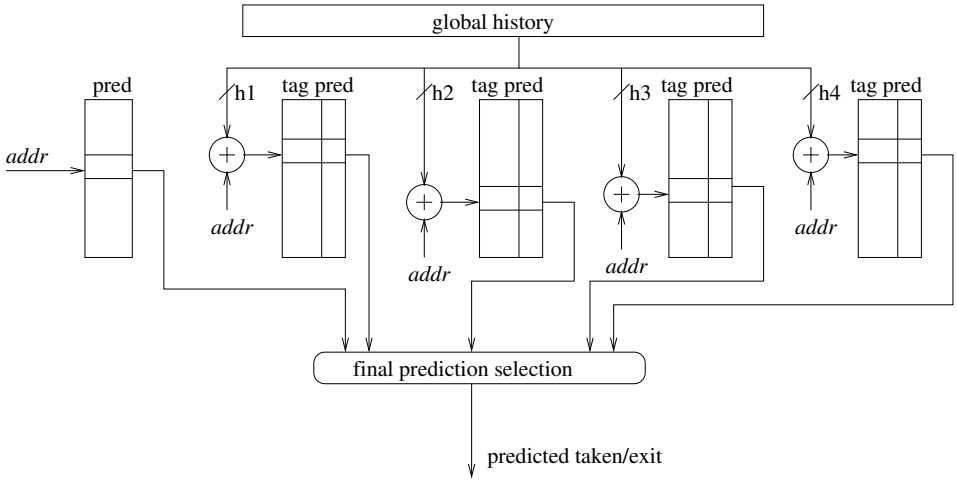


Figure 4.4: A five-component TAGE branch/exit predictor that consists of a bimodal component and four other global+path history indexed components with geometrically increasing history lengths (*h1*, *h2*, *h3*, *h4*). The final direction prediction is made by the longest-history component with a matching tag. If there is no tag match, the bimodal component makes the final prediction.

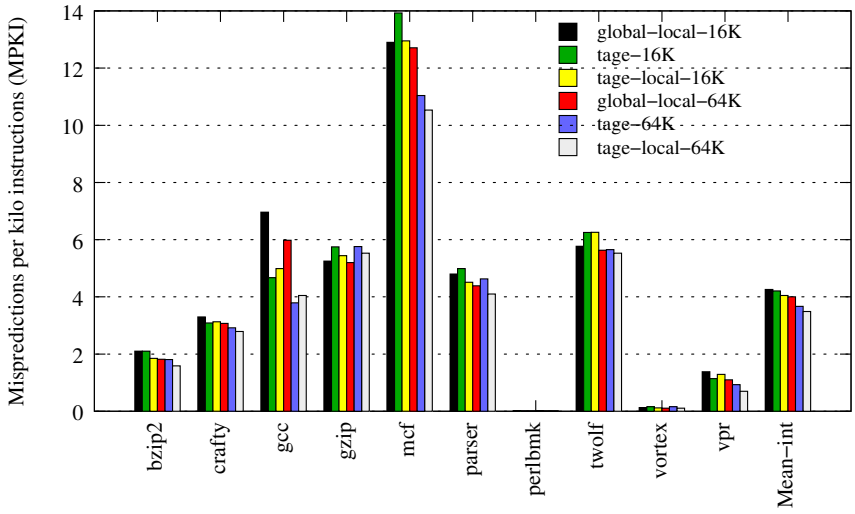
A five-component TAGE branch/exit predictor is shown in Figure 4.4. Mapping a TAGE branch predictor to exit predictions does not require choosers or majority votes as in the GEHL-like exit predictor described previously. It is more straightforward to map the

TAGE predictor to exit prediction compared to mapping the GEHL branch predictor. Our TAGE-like exit predictor's final prediction selection logic is similar to the TAGE branch predictor. One difference in our implementation of TAGE is that we do not use dynamic history or threshold fitting. Another difference is that we use a hash of the path history and the block address bits as the tag in each TAGE table instead of the block address alone. We find that using the history also to calculate the tag achieves slightly lower MPKI compared to using the block address alone (1.5% lower for a 16 KB TAGE predictor). We evaluated different sizes of TAGE predictors with varying number of tables, history lengths, tag lengths, and hysteresis bit lengths.

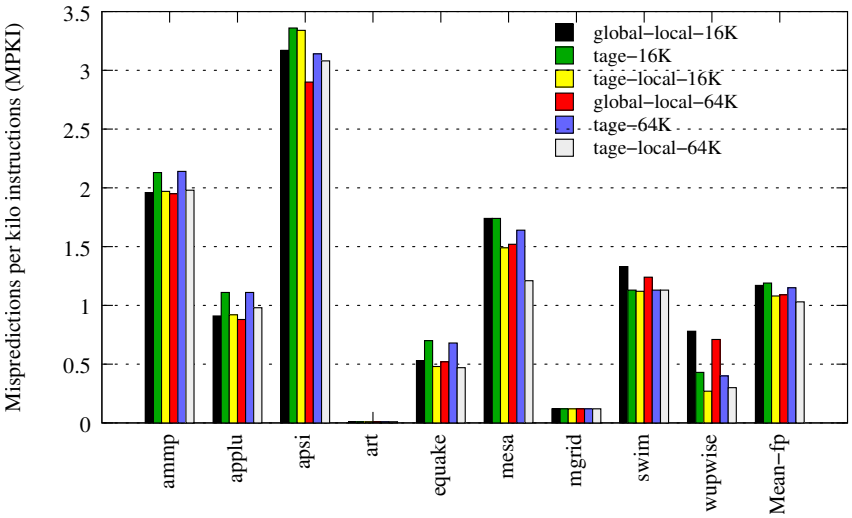
TAGE exit predictor evaluation

Figure 4.5 shows the MPKI achieved by 16 KB and 64 KB TAGE predictors in comparison to the MPKI achieved by a local/global tournament exit predictor. The first bar in the graph represents the 16 KB local/global tournament predictor, the second bar represents the 16 KB TAGE predictor, and the third bar represents the local/TAGE tournament predictor. The improvement of a 16 KB TAGE predictor over a 16 KB local/global tournament predictor is minimal (4.21 MPKI compared to 4.26 for the baseline) for integer benchmarks. For FP benchmarks, the TAGE-like exit predictor achieves marginally higher MPKI (1.19 compared to 1.17 for the baseline) than the local/global tournament predictor. However, for both sets of benchmarks there is a reduction in the MPKI when moving to the local/TAGE tournament predictor (4.05 for integer and 1.08 for FP). The improvement is 4.9% for integer benchmarks and 7.7% for FP benchmarks. For *bzip2*, *parser*, and *vortex* among integer benchmarks, and, *equake*, *mesa*, *swim*, and *wupwise* among FP benchmarks, the local/TAGE tournament is better than the other two predictors. These numbers indicate the relative importance of local predictors when predicting exits even when a sophisticated global-history based predictor like TAGE is employed.

To understand whether there were size limitations for the TAGE-like exit predictor



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.5: MPKI for TAGE-like exit predictors of size 16 KB and 64 KB for integer and FP benchmarks. The first bar shows the MPKI for a local/global tournament predictor, the second bar shows the MPKI for the TAGE predictor, and the third bar shows the MPKI for a local/TAGE tournament predictor, all of size 16 KB. The fourth bar shows the MPKI for a 64 KB local/global tournament predictor, the fifth bar shows the MPKI for a 64 KB TAGE predictor, and the sixth bar shows the MPKI for a 64 KB local/TAGE predictor.

which prevented it from achieving lower MPKI, we simulated several scaled-up configurations. We show one such configuration, scaled-up TAGE-like predictor (in Figure 4.5) and local/TAGE tournament predictor. We compare these predictors with a scaled 64 KB local/global tournament predictor. The fourth bar for each benchmark represents the 64 KB local/global tournament, the fifth bar represents the 64 KB TAGE-like exit predictor while the sixth bar represents the 64 KB local/TAGE tournament predictor. The scaled local/global tournament predictor has an average MPKI of 4.0 for integer programs and 1.1 for FP programs. The 64 KB TAGE predictor offers a good reduction in MPKI for integer programs (8.3%) when compared to the baseline. However for FP programs, the predictor is 5.5% worse, indicating that even at this size, using only global histories is not very effective. The improvements offered by the local/TAGE combination are better: 12.8% for integer and 5.5% for FP compared to the baseline local/global tournament predictor. The mean MPKI of the local/TAGE predictor is 3.5 for integer and 1.0 for FP benchmarks. In summary, for both the 16 KB and the 64 KB exit predictors, the local/TAGE tournament predictor is the best performing predictor.

4.2.3 Perceptron-based neural exit predictor

Perceptrons are simple neurons and are good at predicting binary outcomes using a set of inputs and a set of weights. Perceptrons offer one of the best prediction rates for branch prediction [24, 25]. Perceptron predictors can use global as well as local histories. A set of weights corresponding to the selected branch is used in combination with the history to arrive at the final prediction. Unlike standard two-level predictors, perceptrons do not need 2^N space for N history or index bits. The perceptron tables are indexed using the branch address (or sometimes, in combination with path bits if a two-dimensional array of perceptrons is used) and they only need O(N) space instead of O(2^N). Each entry in the table of perceptrons contains several signed weights. For example, to use a 64-bit history, each entry requires 64 weights. If each weight is an eight-bit signed integer, each entry

requires 512 bits. Even with only 64 entries, the size of the prediction table is 4 KB.

The advantage of a neural predictor is that by using longer histories and tracking highly correlated and poorly correlated branches using eight-bit weights, it offers much better prediction rates. One disadvantage of a perceptron predictor is that the computation of dot products (of each history bit with the corresponding signed weight) takes several cycles. The adder tree computation is O(log N). For branch predictors which need to predict every cycle, ahead pipelining can be employed [23, 24] to hide this latency. Ahead pipelining starts the table access and computation several cycles before the prediction is required using the information available at that time. During successive cycles, when the latest history and address information are available, these parameters are used in selecting the final prediction. Thus, with a slightly reduced accuracy, ahead pipelining can effectively handle long latency predictions. In TRIPS, predictions are made only once every eight cycles for blocks. Hence ahead pipelining may not be required or, if required, ahead pipelining by fewer cycles may be enough.

In prior work [50], Jimenez proposed a neural exit predictor based on local/global perceptron branch predictor. Since exit predictors require one-of-many prediction, this predictor uses several perceptron weight tables, one to predict a subset of exits. For each block, there is a set of perceptrons corresponding to the most frequently seen exits. For example, a block may have eight exits. But if we have only four tables, the four most recently seen exits are stored along with the perceptrons in each of the four entries corresponding to this block in the four tables. The exit in the entry corresponding to the perceptron with the highest excitation (sum of weights) is selected. When a block resolves, the resolved exit is compared to the existing exits stored for this block. When there is no entry for the resolved exit, one of the existing exits is replaced based on LRU replacement. During update, the perceptron weights are updated depending on whether an exit is taken or not. Regular exit histories were used in this multi-perceptron predictor.

Since perceptrons give a signed sum as their final output (the sign of this value in-

dicates taken or not-taken in branch prediction), one way to extend this to exit prediction is by considering eight perceptron tables instead of one table. In this case, unlike the multi-perceptron approach discussed above, storing the corresponding exit and replacement are not needed. Each perceptron table corresponds to one exit in a hyperblock. All the perceptron tables are indexed using block address/path combination. To choose the final exit, we can either use the exit corresponding to the first table prediction "taken" or the exit corresponding to the table which reports the highest sum of weights. We use the highest sum of weights approach. This *8-perceptron* design is straightforward but may waste lot of space in using several tables. It resembles a parallel/multiple branch predictor design.

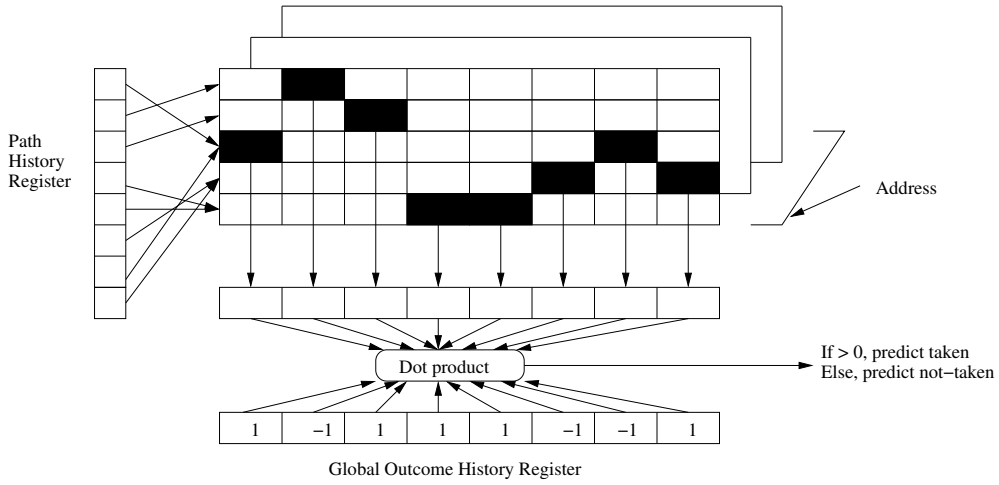


Figure 4.6: Piecewise linear branch predictor consisting of a three-dimensional array of perceptron weights.

We employ the piecewise-linear perceptron predictor [24] to construct the piecewise-linear *8-perceptron* predictor. This predictor is demonstrated in Figure 4.6. For histories of length H, the piecewise-linear perceptron predictor uses a three-dimensional array of perceptrons with *M x N x H* signed weights. One dimension is indexed using block address bits (*log M* bits for *M* addresses). Once indexed, this gives a two-dimensional array (*N x H*) of perceptron weights. Now, for each history outcome from 0th outcome to *(H - 1)*-th

outcome, the corresponding block address (address of the block corresponding to the outcome) bits represent the path bits. These *log N* bits are used to retrieve the weight for the *i*th outcome. Similarly all the history weights corresponding to the block address for each history outcome position are collected and the collective dot product with the outcome bits gives the final weighted sum. The idea is that, every path leading to a block gets a separate set of history weights without much path or block aliasing. Typically, an extra weight called bias weight is maintained for each block. The bias weight indicates the bias of the corresponding exit i.e., whether the exit is predominantly taken or not taken.

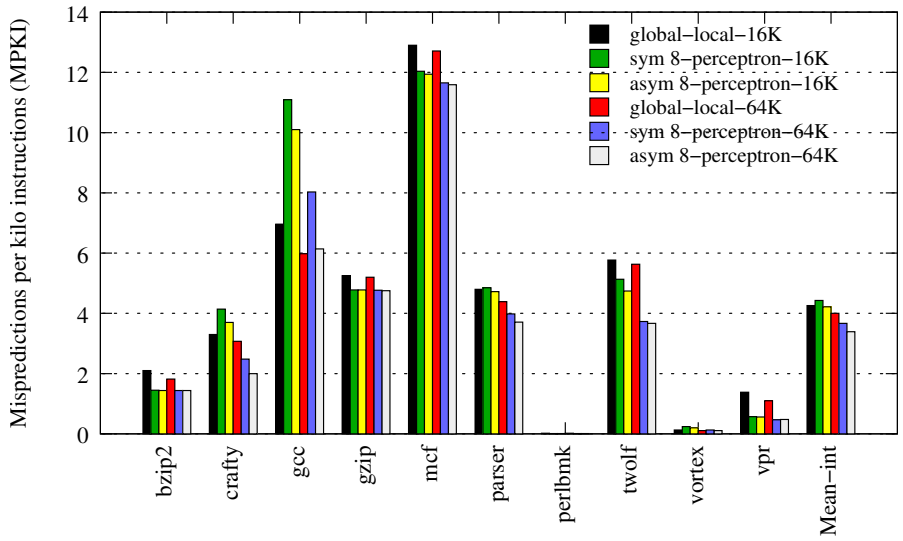
We evaluated several configurations of the piecewise-linear perceptron predictor. When only one path address is used (N is 1), this predictor resembles the global perceptron predictor [25]. When only one block address used (M is 1), this predictor resembles the path-based perceptron predictor [23]. Local history bits may be used in combination with the global history bits [26]. We use L local history bits to compute the sum of local weights separately from a separate array of weights containing L weights in each entry. The remaining global history bits are used in the piecewise-linear configuration as above.

The global and local histories for the perceptron-based predictors can be decoded or encoded exit histories. Encoded exit histories are usual histories used in the predictors described so far (exit bits are encoded using 3 bits). Decoded exit histories represent the taken exit and the other exits in a decoded fashion. For example, if the taken exit ID is *101*, the decoded history would be *00001*. We evaluated encoded and decoded histories with the maximum number of required tables to predict the outcomes of all exit branches in a block (eight tables). The insight behind decoded history bits is that they may provide a more accurate per-branch-level correlation information to be tracked by the perceptron. On the other hand the encoded history uses fewer bits, so if a significantly high percentage of exits taken have higher exit IDs, the encoded version may be better since it offers better history compression. In our evaluations, configurations using decoded exit histories far outperformed those using encoded exit histories.

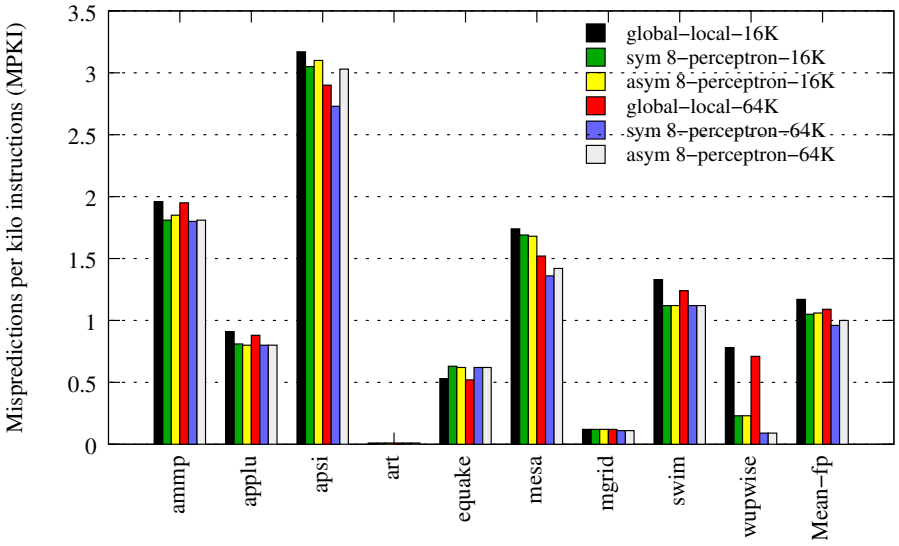
While allocating eight perceptron tables to predict the taken exit, one can allocate same sizes to all tables (symmetric 8-perceptron), or allocate different sizes to different tables based on their frequency of occurrence (asymmetric 8-perceptron). We found that asymmetric allocation is better. More space is allocated to the perceptron table predicting the direction of the first exit, lesser space to the next three tables, and the least space to the next four tables. While varying the number of blocks and number of path addresses, we found that the best piecewise-linear predictors had equal or very close values of M, the number of a blocks addresses and N, the number of path addresses. For most benchmarks, we found that increasing the history length H, while maintaining the size (by decreasing N and M) was beneficial. However, for *gcc*, even though longer histories can help in capturing better correlation, the reduced number of table entries caused more destructive aliasing due to the large number of static blocks.

Piecewise-linear exit predictor evaluation

We show the results of our best piecewise-linear 8-perceptron exit predictor compared to the local/global tournament exit predictor in Figure 4.7. Each predictor is tuned to be approximately 16 KB in size. The symmetric perceptron predictor performs slightly worse than the local/global tournament predictor while the asymmetric perceptron predictor performs better than the local/global tournament predictor. We also show a scaled-up 64 KB 8-perceptron exit predictor compared to a scaled-up tournament predictor. The 16 KB symmetric 8-perceptron predictor achieves an MPKI of 4.43 compared to the 4.26 MPKI achieved by the local/global tournament predictor for integer benchmarks. However, the 16 KB asymmetric 8-perceptron predictor performs slightly better with an MPKI of 4.22. For the 64 KB sizes evaluated on integer benchmarks, the best predictor is again the asymmetric 8-perceptron predictor with an MPKI of 3.39. The symmetric perceptron is also better than the tournament predictor with an MPKI of 3.67. The local/global tournament predictor achieves an MPKI of 4.0. Hence, for 64 KB predictors, the symmetric 8-perceptron



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.7: MPKI for piecewise-linear perceptron-inspired 8-perceptron exit predictors of size 16 KB and 64 KB for integer and FP benchmarks. The first bar shows the MPKI for a local/global tournament predictor, the second bar shows the MPKI for a symmetric 8-perceptron predictor, and third bar shows the MPKI for an asymmetric 8-perceptron predictor, all of size 16 KB. The fourth bar shows the MPKI for a 64 KB local/global tournament predictor, the fifth bar shows the MPKI for a symmetric 8-perceptron predictor, and the sixth bar shows the MPKI for an asymmetric perceptron predictor.

predictor achieves 8.3% reduction in MPKI while the asymmetric 8-perceptron predictor achieves 15.3% reduction in MPKI compared to the 64 KB local/global predictor.

For FP benchmarks, the symmetric predictor is the best performing predictor with an MPKI of 1.05 for the 16 KB configuration (compared to 1.06 for asymmetric and 1.17 for tournament). For a 64 KB size, the symmetric predictor achieves 0.96 while the asymmetric predictor achieves 1.00 and the tournament predictor achieves 1.09. For the FP suite, the fraction of time exits with ID 4 and higher are encountered is high for several benchmarks as shown in Chapter 2. Thus, using smaller perceptron tables for exits with IDs 4 and higher lowers the overall prediction accuracy of the FP suite.

4.3 Exit predictors using efficient binary predictors: The PPE predictor

State-of-the-art branch predictors have been reported to have high accuracies of more than 95% [26, 75]. However, when mapped to exit prediction, their misprediction rates are much higher. A possible reason is the inefficient use of space due to the use of three bits for exits (when several blocks have only two to four frequently taken exits). Another reason may be the inherent difficulty in predicting exits (1 of 8) compared to predicting branches (which is a binary prediction). Furthermore, exits approximate the predicate path inside the block but do not represent the path accurately. Some branches in the basic block code converted to predicates in the hyperblock code which lie on the path leading to an exit are not represented by the exit numbers. Hence the correlation information from the predicated branches is not available. This aspect is explored in detail in Chapter 5. Finally, a design/implementation drawback arises when mapping branch predictors to exit prediction. Several sources of inefficiencies are introduced like the use of a chooser instead of an adder in the GEHL-like exit predictor and the use of eight tables in the perceptron-based exit predictor (leading to a reduction in the per-table prediction state). So far, the predictors we evaluated in this chap-

ter, do not perform significantly better than the local/global tournament exit predictor for a size of 16 KB. However, if a larger predictor is considered (64 KB), the results are more impressive. Exit predictors like the local/TAGE tournament predictor and the 8-perceptron asymmetric predictor perform better than the local/global tournament predictor.

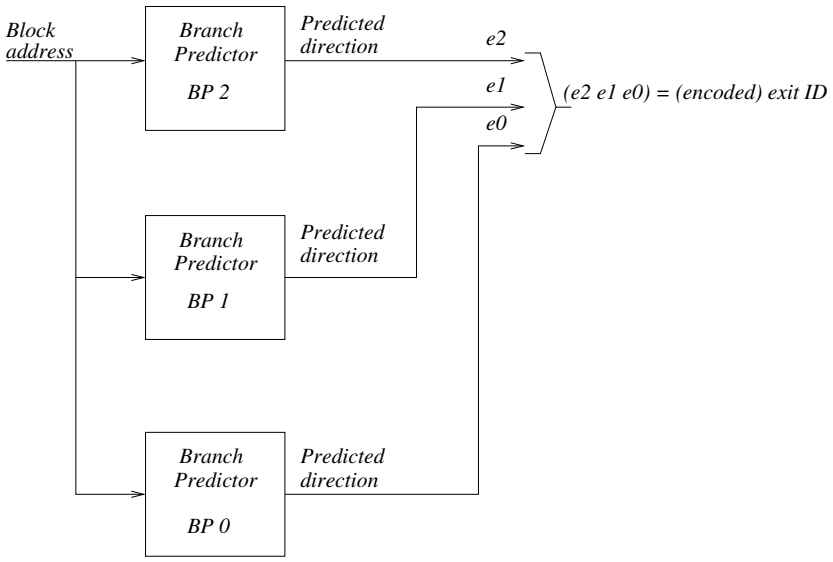


Figure 4.8: Exit Predictor using a Post-Prediction Encoding (PPE) Design which uses three predictors, one predictor to predict each bit of the three-bit exit ID.

In this section, we explore the Post-Prediction Encoding (PPE) exit predictor. The idea stems from the fact that perceptrons and other state-of-the-art branch predictors are good at predicting one of two values/sets (binary prediction) and instead of trying to use a binary prediction for each exit and choosing one among eight, we can use a binary prediction for each bit of the exit thereby reducing the number of predictors to *log(maximum exits in a block)* which is only three in our case. Each component predictor predicts one of the three bits in an exit ID. The final result is the concatenation of the predictions of the three components which gives the encoded exit. We can also take advantage of the fact that most taken exits have exit ID 4 or lower. Hence it may be useful to size the predictors for each of the components differently. A lower size for the predictor predicting the MSB bit of an exit

may not be very inefficient. A block diagram of a generic PPE predictor which can use any branch predictor for each of its components is shown in Figure 4.8.

Each component predictor, in effect, chooses which of two sets of exits (each set containing four exit IDs) will have the taken exit. A disadvantage with using a separate prediction for each bit is that when regular two-level predictors are used, the hysteresis bits take up more space in this predictor as they are used for each table. Hysteresis bits could be shared, but it may be difficult to share bits among components that have different sizes or when the components are of different types. The PPE predictor can use any binary control flow predictor as a generic *0-or-1* predictor. The main goal of the PPE predictor is to leverage the improvements in branch direction (binary) predictors to design efficient aggregate control flow predictors. Note that the PPE predictor scales much better than multiple branch predictors. If the maximum number of exit IDs allowed is *N*, it uses only *O(log N)* predictors instead of *O(N)* predictors present in regular multiple-branch predictors.

The PPE predictor can use various types of branch predictors as its components to predict each bit. The history used can be the history representing the previous values of the corresponding exit bit (i.e., only the most significant bit, MSB of an exit ID if the predictor is predicting the MSB), or an entire exit history just as in previously described exit predictors. We found that using just the previous values of the corresponding exit bit leads to severe loss in correlation and reduced accuracy. This is because even though we try to use three branch predictors to predict the three bits in parallel, we are in effect trying to predict a single exit and the exit that is taken may depend on various bits in the complete history of all exit bits, and not just the corresponding exit bit. Hence we find that using the full exit history is better. Of course, the full exit history may also comprise of truncated exit bits to be able to use more exits in the history just like in regular local or global exit predictors.

We demonstrate the PPE prediction scheme using the piecewise-linear perceptron predictor as the component predictor. We choose the neural predictor because this predictor

is good at predicting branches but it is difficult to map efficiently to exit prediction and needs to use several tables to make predictions as in the 8-perceptron exit predictor. In the last section, we saw that for a 16 KB size, this predictor was only marginally better than the local/global tournament predictor. In this section, we explore the PPE version of the piecewise-linear predictor. We use three components to predict the three bits of an exit. We found that using three different sizes for the three components typically gave higher prediction accuracy than using components of the same size. We evaluate three types of PPE predictors as described below:

1. **Symmetric PPE**: In a symmetric PPE predictor, all the three components have the same size and the component predictors always predict the same bit positions i.e., component 0 always predicts the LSB (bit 0 of the exit ID), component 1 always predicts the middle bit (bit 1), and component 2 always predicts the MSB (bit 2).

2. **Symmetric rotating PPE**: In a symmetric rotating PPE predictor, all the three components have the same size and the component predictors do not always predict the same bit positions. For example, component 0 can predict the LSB for some hyperblocks while predicting the MSB or the middle bit for other hyperblocks. To implement this, we choose a static hash of the block address which can give one of three values: 0, 1 or 2. Depending on the number, the components that should predict each bit are assigned. For example, if the hashed output is 0, we can assign component 0 to predict MSB, while component 1 predicts the LSB, and component 2 predicts the middle bit. This predictor while retaining the same-size component of the symmetric PPE, can offer a better utilization of the tables by partitioning each of the bit predictions across the three components (for example, not penalizing the prediction of a difficult-to-predict LSB bit that may need more space than an easy-to-predict MSB bit).

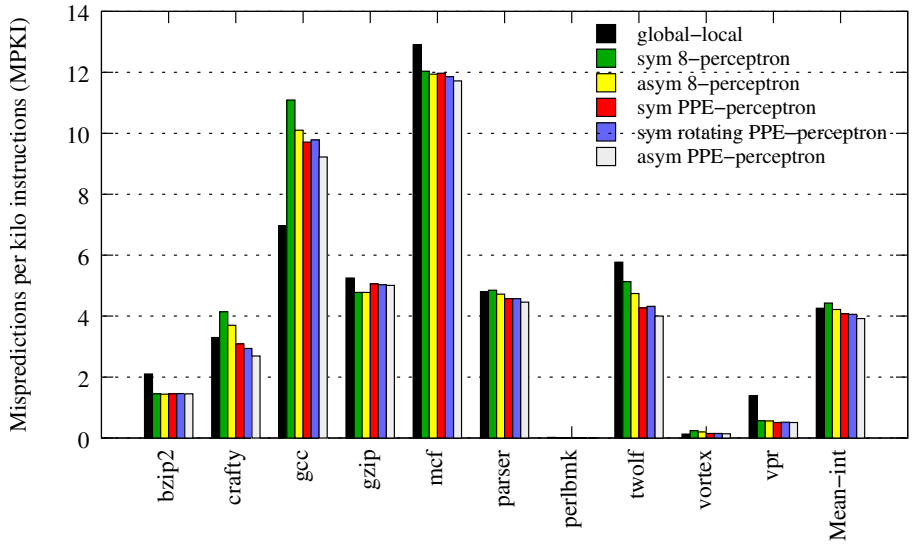
3. **Asymmetric PPE**: In an asymmetric PPE predictor, the three components need not

have the same sizes. Like the symmetric PPE predictor, this predictor also uses fixed assignments of the component predictors to exit bit positions for predictions. The advantage of an asymmetric PPE over the symmetric PPE is that each of the tables can be sized based on the variability and frequency of the taken exit IDs. If most of the taken exit IDs lie between 0 and 3, the MSB component can be smaller than the other components since most of the exit MSB values will be 0.

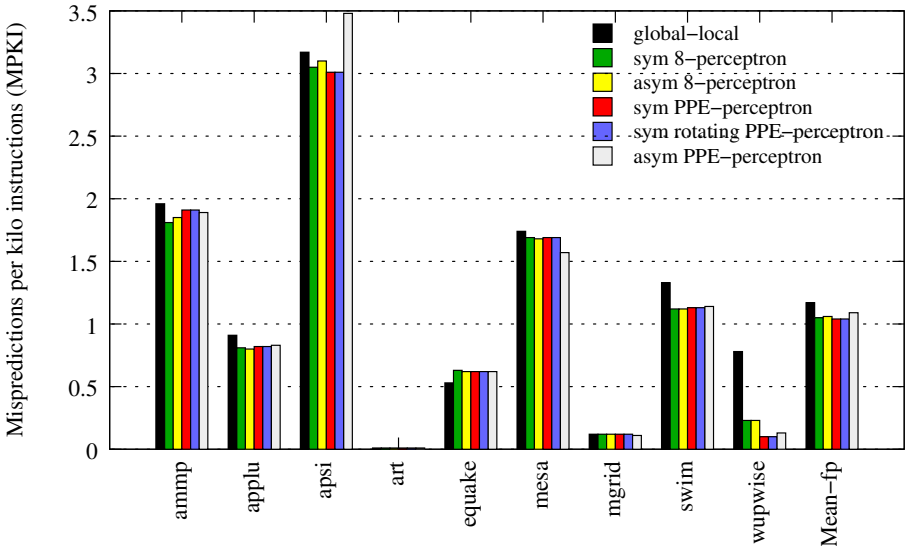
PPE exit predictor evaluation

Figure 4.9 shows the exit MPKI for a 16 KB symmetric, rotating symmetric, and asymmetric PPE predictor using piecewise-linear branch predictor based components. Figure 4.10 shows the comparison for 64 KB predictors. We compare the PPE predictor with the local/global tournament predictor and the symmetric and asymmetric 8-perceptron predictor described in the previous section. The first bar shows the local/global tournament predictor, the second bar shows the asymmetric 8-perceptron predictor, and the third bar shows the symmetric 8-perceptron predictor. The fourth bar shows the symmetric PPE perceptron predictor, the fifth bar shows the symmetric rotating PPE perceptron predictor, and the final bar shows the asymmetric PPE perceptron predictor. For the asymmetric PPE predictor, the component predicting the MSB was given the smallest size (22% of the total size) and the component predicting the LSB was given the largest size (45% of the total size).

For the 16 KB configuration, the mean integer MPKI values are 4.26 for local/global, 4.43 for symmetric 8-perceptron, 4.22 for asymmetric 8-perceptron, 4.08 for symmetric PPE perceptron, 4.06 for symmetric rotating PPE perceptron, and 3.92 for asymmetric PPE perceptron. Among 16 KB predictors, the best predictor for the integer suite is the asymmetric PPE with an average MPKI of 3.92. This is the best 16 KB exit predictor among the predictors we evaluated in this chapter. The asymmetric PPE predictor is able to predict well for difficult-to-predict benchmarks like *mcf*, *parser*, and *twolf*. The symmetric PPE predictor with an MPKI of 4.08 and the symmetric rotating PPE predictor with an MPKI

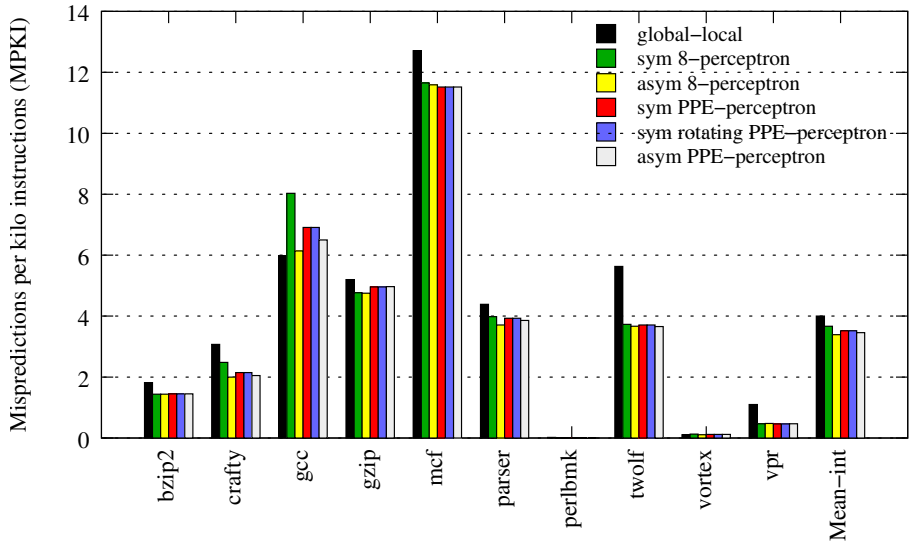


(a) SPEC2K integer benchmarks

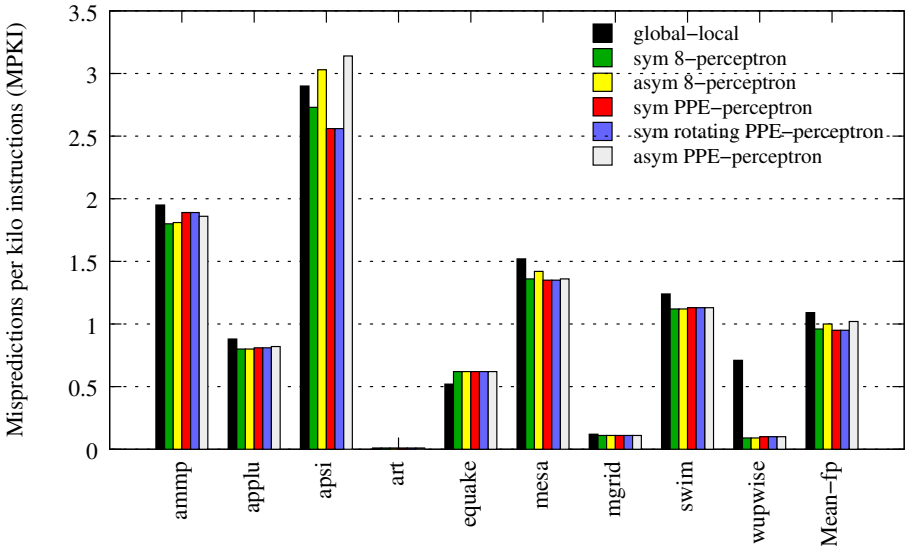


(b) SPEC2K FP benchmarks

Figure 4.9: MPKI for SPEC integer and FP benchmarks for 16 KB PPE exit predictors with the piecewise-linear exit predictor as the component predictor for its three components. The first bar shows the MPKI for a local/global tournament predictor, the second bar shows the MPKI for a symmetric 8-perceptron predictor, and third bar shows the MPKI for an asymmetric 8-perceptron predictor. The fourth bar shows the MPKI for a symmetric PPE perceptron predictor, the fifth bar shows the MPKI for a symmetric rotating PPE perceptron predictor, and the sixth bar shows the MPKI for an asymmetric PPE perceptron predictor. All predictors are approximately 16 KB.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.10: MPKI for SPEC integer and FP benchmarks for 64 KB PPE exit predictors with the piecewise-linear exit predictor as the component predictor for its three components. The first bar shows the MPKI for a local/global tournament predictor, the second bar shows the MPKI for a symmetric 8-perceptron predictor, and third bar shows the MPKI for an asymmetric 8-perceptron predictor. The fourth bar shows the MPKI for a symmetric PPE perceptron predictor, the fifth bar shows the MPKI for a symmetric rotating PPE perceptron predictor, and the sixth bar shows the MPKI for an asymmetric PPE perceptron predictor. All predictors are approximately 64 KB.

of 4.06 perform better than the local/global and 8-perceptron predictors. Compared to the asymmetric 8-perceptron, the symmetric PPE predictor is 3.3% better while the asymmetric PPE predictor is 7.1% better. The symmetric rotating PPE has a slightly lower MPKI compared to the regular symmetric PPE but it is still worse than the asymmetric design. The rotation provides some distribution but it still does not result in effective space utilization compared to the asymmetric PPE.

For the FP benchmarks, there is hardly any improvement while moving to a PPE design from the 8-perceptron design. The best predictors are the symmetric PPE predictor and the symmetric rotating PPE predictor. Both these predictors have an MPKI of 1.04 while the asymmetric predictor achieves an MPKI of 1.09. The symmetric 8-perceptron predictor is the second-best predictor with an MPKI of 1.05 while the asymmetric 8-perceptron predictor has an MPKI of 1.06. The global/local tournament predictor has an MPKI of 1.17.

For the 64 KB configuration, the results are somewhat disappointing. The mean integer MPKI values are 4.0 for local/global, 3.67 for symmetric 8-perceptron, 3.39 for asymmetric 8-perceptron, 3.52 for symmetric PPE perceptron, 3.52 for symmetric rotating PPE perceptron and 3.46 for asymmetric PPE perceptron. The best PPE predictor for 64 KB is the asymmetric predictor with an MPKI of 3.46 which still falls below the 3.39 MPKI achieved by an asymmetric 8-perceptron by about 2.1%. Since a 64 KB predictor is quite large, the 8-perceptron does not suffer due to the use of eight tables. For the FP benchmarks, the results show a trend similar to the 16 KB predictors. The symmetric PPE and the symmetric rotating PPE achieve the same MPKI of 0.95, which is the lowest among all the predictors compared. The asymmetric PPE predictor achieves an MPKI of 1.02. The symmetric and asymmetric 8-perceptron predictors have MPKI values of 0.96 and 1.0 respectively. The local/global tournament predictor has an MPKI of 1.09. The asymmetric 8-perceptron predictor and the symmetric PPE predictors achieve approximately same MPKI values.

On the whole, we observe that the PPE predictor, especially the asymmetric PPE

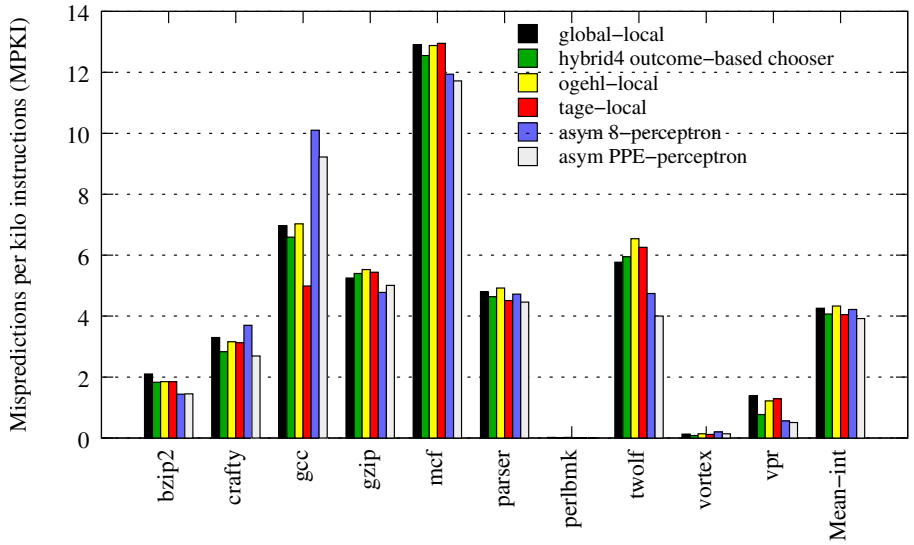
predictor, can provide significant improvements for integer benchmarks when considering small and medium sized predictors. For floating-point benchmarks, the PPE and the 8-perceptron predictors perform almost on par with each other. Among the PPE predictors the symmetric and symmetric rotating PPE predictors perform better for the FP benchmarks while the asymmetric PPE predictor is the clear winner for integer benchmarks. The PPE predictor shows promise for future exit/multi-bit predictions. The design can be potentially improved by tuning the predictors further, using different types of predictors to predict each bit and sharing hysteresis bits across predictor tables.

4.4 Comparison of exit predictors

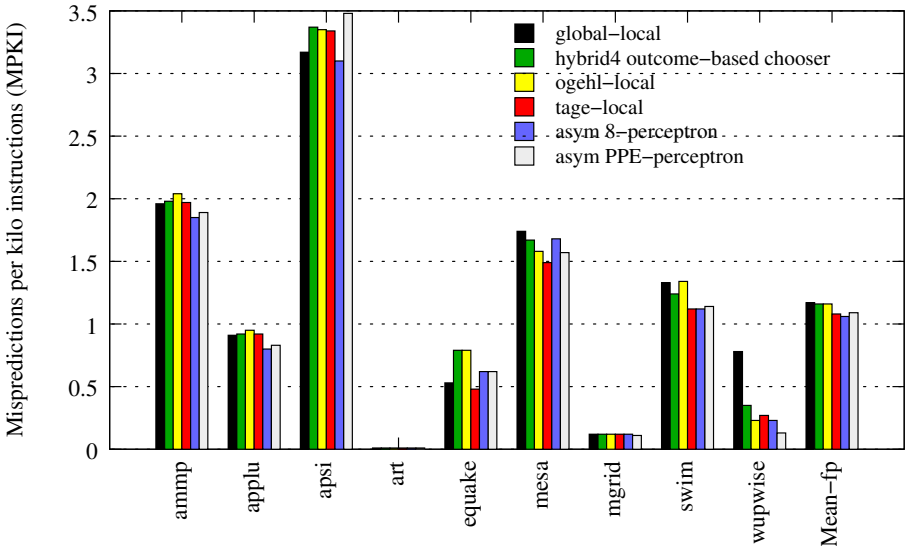
In this section, we compare the best exit predictors described in the previous sections in this chapter. For each type of exit predictor described previously, we choose the best performing predictor discussed earlier. All the predictors are of size 16 KB (approximately, with a 15% overhead allowed). The predictors are compared to a baseline 16 KB local/global tournament exit predictor.

Figure 4.11 shows the exit MPKI for six exit predictors for SPEC integer and floating-point benchmarks. The first bar for each benchmark shows the exit MPKI of the local/global tournament predictor. The second bar shows the exit MPKI of the hybrid-4 predictor with the component outcome and history-based chooser (*C-OUTCOME*). The third and fourth bars show the exit MPKI values of the local/OGEHL tournament predictor and the local/TAGE tournament predictor respectively. The fifth and final bars show the exit MPKI values of the asymmetric 8-perceptron predictor and the asymmetric PPE perceptron predictor respectively.

For integer benchmarks, the mean exit MPKI values of the six predictors in order (as in the figure) are 4.26, 4.07, 4.33, 4.05, 4.22, and 3.92. The worst performing predictor is the local/OGEHL predictor with an MPKI of 4.33. The local/global tournament predictor performs slightly better with an MPKI of 4.26. The asymmetric 8-perceptron predictor



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.11: Comparison of exit MPKIs from the best 16 KB exit predictors for SPEC integer and FP benchmarks. The first bar shows the MPKI for the local/global tournament predictor, the second bar shows the MPKI for the hybrid-4 (bimodal/local/global/path hybrid) predictor with outcome-based chooser, the third bar shows the MPKI for the local/OGEHL tournament predictor, the fourth bar shows the MPKI for the local/TAGE tournament predictor, the fifth bar shows the MPKI for the asymmetric 8-perceptron neural predictor, and the final bar shows the MPKI for the asymmetric PPE perceptron predictor.

performs slightly better than the local/global tournament with an MPKI of 4.22. The hybrid-4 predictor and the local/TAGE predictor perform almost on par with each other (MPKI values of 4.07 and 4.05 respectively) and offer about 4.7% improvement in MPKI over the local/global tournament predictor. The best 16 KB exit predictor is the asymmetric PPE predictor with an 8% improvement in MPKI over the local/global tournament predictor. For benchmarks like *bzip2*, *crafty*, *mcf*, *parser*, and *twolf* the asymmetric PPE predictor gives the lowest MPKI. However, for a difficult-to-predict benchmark like *gcc*, it performs significantly worse (with an MPKI of 9.22) than the tournament predictor. The reason for the poor predictability for *gcc* for both the asymmetric PPE that uses perceptron predictor components and the 8-perceptron predictor is that *gcc* has a large number of static blocks and there is severe capacity aliasing in the perceptron tables. Since a perceptron predictor uses several bits in each table entry, the number of table entries has to be kept low. This is in contrast to regular two-level predictors where the prediction table (level 2) entries contain only four bits typically (exit ID + hysteresis bit). When considering a 64 KB PPE predictor (Figure 4.10), the asymmetric PPE predictor performs much better and is only slightly worse than the 64 KB local/global tournament predictor for *gcc*. The best predictor for *gcc* is the local/TAGE tournament predictor with an MPKI of 4.99.

For the FP benchmarks, the mean exit MPKI values of the six predictors in order (as in the figure) are 1.17, 1.16, 1.16, 1.08, 1.06, and 1.09. The worst performing predictor is the local/global tournament predictor. The hybrid-4 predictor and the local/OGEHL tournament predictor have almost the same MPKI (1.16) as the local/global tournament. The asymmetric PPE perceptron and the local/TAGE tournament predictor have almost the same MPKI values (1.09 and 1.08). The best predictor for the FP suite is the asymmetric 8-perceptron predictor with an MPKI of 1.06 which indicates an improvement 9.4% over the local/global tournament predictor.

Considering both benchmarks suites, the best overall predictor is the asymmetric PPE predictor that shows an improvement of 8% for the SPEC integer suite and an improve-

ment of 6.8% for the SPEC FP suite when compared to the 16 KB local/global tournament predictor.

4.5 Designing indirect branch predictors to improve target prediction

Indirect branch prediction is an important component of target predictors. Typically there are several indirect branches and calls in integer programs. In C++ and Java programs the number of indirect branches and calls is high because of object-oriented programming and virtualization.

Indirect branch prediction is difficult when the number of dynamically taken targets for a static indirect branch is high. Indirect branches are also difficult to predict when the targets are not correlated properly with the program path. Several mechanisms have been proposed for indirect branch prediction which use global and path histories [5, 9], filtering-based multi-stage predictors [10], and tagged tables with geometric history lengths [62]. Kim et al [33] do not employ a separate indirect branch predictor. They use the direction predictor to predict indirect branches also by converting each indirect branch into several virtual direct branches and predicting them successively until a virtual branch is taken.

To store indirect branch targets, we can use the BTB (shared with direct branches) or use a separate BTB-like structure (we call this IBTB) for indirect branches alone. In our study, we use the IBTB to predict indirect branches and calls. Since indirect branches and calls can have targets far away from the current block, we store absolute target addresses in a separate table called the IBTB. The BTB stores only offsets for direct branches. To support accurate return address prediction corresponding to returns in functions called by indirect calls, we need to learn the return addresses as in the CTB. Instead of storing return addresses also in the IBTB, we store them in the CTB itself. Hence in our mechanism, the CTB stores the return addresses corresponding to the indirect calls while the IBTB stores

the indirect call targets.

Throughout this section, we use the same predictor structure in all the experiments other than the indirect branch predictor. The exit predictor is a 16 KB local/global tournament predictor while the target predictor is a 16 KB multi-component predictor consisting of the Btype, BTB, CTB, and RAS. We use an "improved" target predictor that uses longer branch offset widths in the BTB and longer return offsets in the CTB as described in Chapter 3. We evaluate a separate indirect branch predictor structure as part of the multi-component target prediction. To support indirect branch prediction, we change the number of types predicted by the Btype predictor to six (from four). The four basic types are branch, call, return, and sequential branch. The two new types are indirect branch and indirect call. The Btype predictor needs to have one extra bit in each entry to predict these six types.

Once the structure to store the indirect branch targets is finalized, the different ways of indirect branch prediction we evaluate can be considered to be different ways of indexing the IBTB table. One of the simplest ways to predict indirect branches is to index the IBTB like the BTB is indexed, using a hash of the block address and the predicted exit. We call this the address-based predictor. This is a simple predictor but achieves almost the same MPKI as a block predictor without an indirect branch predictor (since the BTB does not suffer very much from aliasing, aliasing benefits are also not present when using the BTB to predict indirect branch).

Most of the prior work in indirect branch prediction has considered using global or path histories with single or multiple tables (hybrid prediction). We evaluate global history based indirect branch predictors which use the global exit history in combination with the block address to index into the IBTB and retrieve the predicted target. We also explore path history based predictors which employ path histories (series of block address in the path leading to the current block) instead of global histories. We also evaluate a variation of the global history based predictor, in which we concatenate the predicted exit along with the global history to index into the IBTB table. We expect that this may offer better resolution

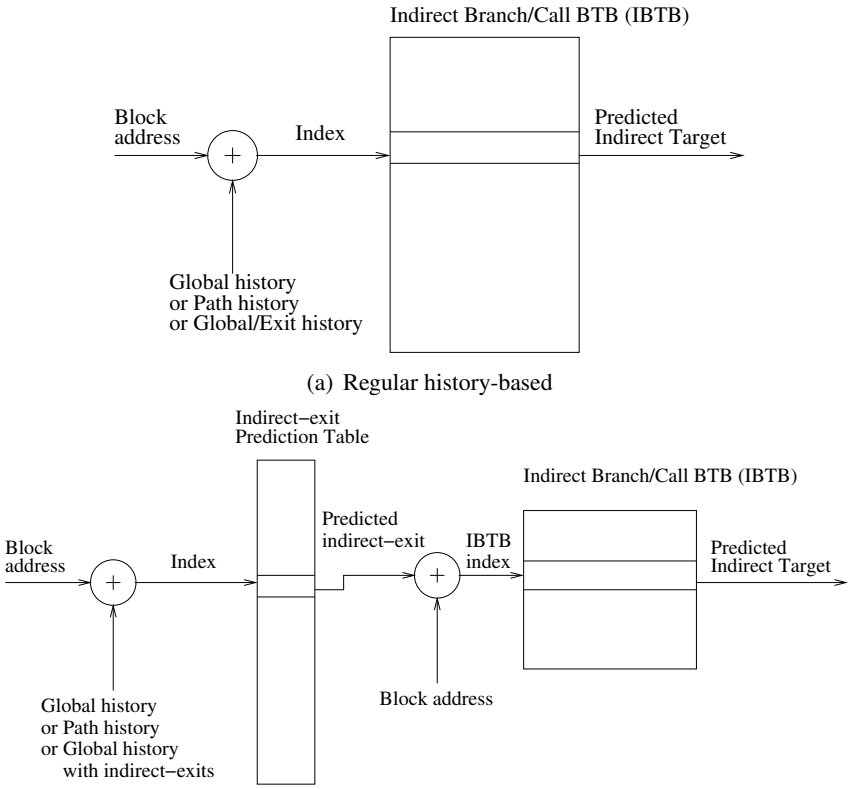
due to the presence of the exit bits in the index.

4.5.1 Indirect branch predictors inspired from exit predictors

Several proposed indirect branch predictors use the history and block address combination to index into the IBTB table. The IBTB table requires large sizes or tags with set-associative entries to reduce aliasing and provide highly accurate indirect predictions. When hybrid predictors are used for difficult indirect branch prediction, each predictor needs to have an IBTB table which is area- and power-expensive. This problem is similar to the exit followed by target or direct target prediction we faced when considering block predictors for TRIPS. Using an exit predictor followed a target predictor, in a two-stage prediction offers higher accuracies because of more efficient use of space. With hybrid techniques, each exponential-sized table need only contain few exit bits and the targets can be kept in simple tables in the target predictor. These target tables are indexed using the block address and the exit bits.

One similarity between a hyperblock and an indirect branch is that both can have multiple targets. But at any point of time, exactly one target is reached. This target is reached through an exit in a hyperblock. Only this exit fires, the rest of the exits do not fire. Similar to the hyperblock exit, we can think of an indirect branch as having several exits (as many as it has static or dynamic targets) and exactly one of the exit firing to reach a specific target. Though there is no exit encoding in the indirect branch, if we are able to generate an encoding dynamically, we can use a two-stage prediction mechanism as for the exit predictor described above. In the first stage we use the block address and predicted hyperblock exit to index into prediction tables to generate the indirect-exit identifier. In the second stage, we use the block address and this indirect-exit ID to predict the indirect target. Just like how the BTB is indexed using the block address and hyperblock exit ID, the IBTB is indexed using the block address and the indirect-exit ID.

The regular history-based indirect branch predictor and the two-stage indirect-exit



(a) Regular history-based

(b) Indirect-exit and history based

Figure 4.12: Comparison of history-based indirect branch predictor and two-stage indirect-exit based indirect branch predictor inspired from exit predictors.

based predictor designs are compared in Figure 4.12. We implemented three types of the two-stage indirect prediction as listed below.

- Simple global exit history-based prediction for indirect-exit that uses a single level-2 table to store indirect-exits.
- Simple path history-based prediction for indirect-exit that uses a single level-2 table to store indirect-exits.
- Global exit history-based prediction including the indirect-exits also in the history along with regular hyperblock exits.

The two simple history based schemes using global and path history resemble global and path-based exit predictors. The third predictor uses global exit history along with the indirect-exits corresponding to indirect targets which helps represent the indirect branch's outcome concisely in the history along with the global exit bits. Adding relevant indirect branch correlation may help in the prediction of future indirect-exits. In a regular path-history based indirect branch predictor, the indirect outcome is also captured because we use block address bits in the history. However in the global-history based predictor, only the hyperblock exits are encoded and if an exit is from an indirect branch it is as effectively expressed. We can consider hybrid schemes as well as more sophisticated schemes to predict the indirect-exit. The IBTB can be made small and tagged, but we use a tag-less IBTB in all the indirect-exit-indexed predictors.

Prediction and updates

During prediction, the history is read and then the second-level indirect exit prediction table is read using a hash of the block address and the history to retrieve the predicted indirect-exit. Using the block address and the predicted indirect-exit, the IBTB is read to retrieve the indirect target address. Speculative updates are required for the global or path histories as done for the history-based hyperblock exit predictors.

A difficult component of this indirect-exit based predictor is in allocating appropriate indirect-exit IDs to targets of indirect branches and updating the indirect-exit prediction table at commit time. The IDs can be global, i.e., every indirect branch target in the program gets a unique ID or local, i.e., every target of an indirect branch gets a unique ID but IDs can be the same across targets of different indirect branches. We choose the local scheme for our experiments since it requires fewer bits in the exit prediction table entries.

Indirect-exit identifier generation

An indirect-exit ID can be allocated by tracking the targets for each indirect branch and allocating a new ID when a previously unseen target is encountered at commit time. This may involve some complexity of tracking and comparing target addresses. However this scheme has the advantage of providing accurate non-aliased identifiers for each unique target of every indirect branch (provided the maximum number of targets is not more than the number of targets supported in the table). Another way to generate IDs is using compiler hints to indicate the IDs for each target address if the compiler knows a few possible target addresses at compile time using profile-driven or static analysis.

Instead of using the above techniques, we choose a simple dynamic scheme to generate the indirect-exit ID. We use a few lower order bits of the indirect target address as the indirect-exit ID. This technique is extremely fast and simple and requires no extra storage but it may result in aliasing of different targets to the same ID. In our evaluations we found this aliasing to be minimal when compared to perfect ID generation.

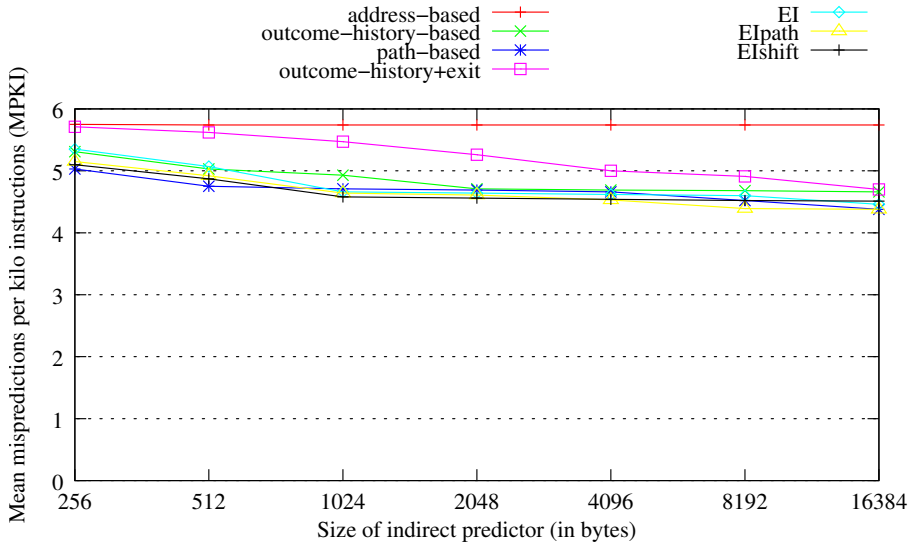
4.5.2 Indirect branch predictor evaluation

We now evaluate a simple address-based indirect branch predictor, global and path history-based indirect branch predictors, global-history and exit combination predictor, exit-predictor-inspired global-history predictor, and exit-inspired indirect branch predictors. For all the history-based predictors, we evaluated various exit bit widths and history widths. For the exit-inspired indirect branch predictors, we varied several parameters like exit bit widths, history widths, level-2 indirect-exit prediction table size and width, and width of indirect-exits in history. We evaluate seven predictor sizes ranging from 256 bytes to 16 KB. As described earlier in this section, the exit predictor is 16 KB and the rest of the target predictor is 16 KB in size. When calculating the total target predictor size we should include the indirect predictor size also. The other modification is the addition of one bit to each entry of the Btype predictor to be able to predict two more types. For a given size in the

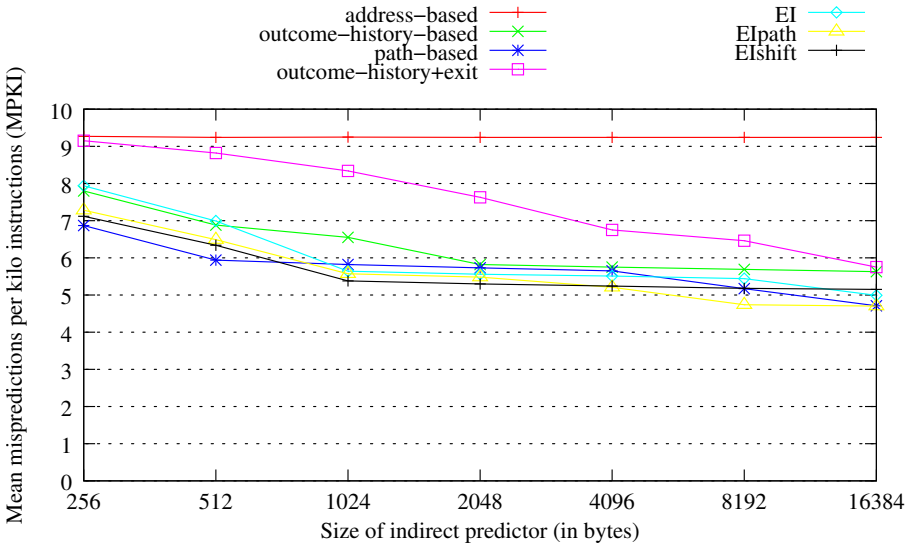
graphs, all address-based and history-based predictors have the exact size while the exit-inspired predictor is at most the same size. This is because we always allocate half the size of the IBTB as in the other predictors for the exit-inspired predictors and the remaining half is not always fully allocated to the level-2 indirect-exit prediction table (depending on the indirect-exit ID width and the number of entries).

Figure 4.13 shows the overall MPKI for the seven different predictors for all integer benchmarks and for a subset of three integer benchmarks (*crafty*, *gcc*, *perlbmk*). We do not evaluate the FP benchmarks as they have almost 0% indirect branches/calls. We show the mean integer MPKI varying over the indirect predictor size as well as the mean integer MPKI of three benchmarks which have significant number of indirect branches or calls. These benchmarks are *gcc*, *perlbmk*, and *crafty*. All the other benchmarks have insignificant numbers of indirect control instructions. The overall mean MPKI shows an interesting trend. The address-based predictor has hardly any improvement when scaling the indirect branch predictor (5.75 to 5.74 MPKI). But all the other indirect predictors which are history-based have reduction in their overall MPKI values when the indirect predictor is scaled.

The best predictor at each size varies. For the smallest predictors with sizes of 256 and 512 bytes, the best predictor is the regular path-history based predictor. This predictor achieves an overall MPKI of 5.03 with 256 bytes and 4.75 with 512 bytes of storage. The exit-inspired (EI) predictors are unable to predict effectively with a very small IBTB (the IBTB size in any EI predictor is only half the size of the IBTB in the regular predictors) which cannot hold many targets. When the indirect predictor size is increased to 1024 bytes or 1 KB, the best predictor is the EIshift predictor with an MPKI of 4.58. This is the global exit-history based indirect-exit predictor with indirect-exits shifted into the exit history. This predictor is the best in the 2 KB size also (with an MPKI of 4.56). In the next two sizes, 4 KB and 8 KB, the best predictor is the EIpath predictor which is a path history-based indirect-exit prediction. It achieves an MPKI of 4.53 and 4.39 with 4 KB and 8 KB respectively. The EIpath and the EIshift predictors are very close in the 4 KB category. The



(a) All SPEC integer 2000



(b) Crafty, Gcc, Perlbmk

Figure 4.13: Overall mean MPKI and overall subset mean MPKI for seven different indirect branch predictors for sizes ranging from 256 bytes to 16 KB. The exit predictor is a 16 KB local/global tournament predictor. The rest of the target predictor is a 16 KB improved target predictor that uses longer offsets in the BTB and the CTB. Address-based (*address-based*), Global exit history-based (*outcome-history-based*), Global exit history based including hyperblock exit (*outcome-history+exit*), Path history-based (*path-based*), Exit-inspired with global history (*EI*), exit-inspired with path history (*EIpath*), and exit-inspired with global history including indirect-exit shifted into history (*EIshift*) are shown.

path-based and EIshift predictors achieve an MPKI of 4.52 in the 8 KB category. For the 16 KB configuration, the best predictors are the regular path-based predictor and the EIpath predictor with an MPKI of 4.38 for both.

The results from the global-history based indirect predictor are close to the path-history based predictor for many sizes. Similarly, the results from the EI predictor which is global-history based are slightly inferior to the results from the EIpath predictor. The MPKI numbers for the global history and resolved-exit combination history predictor (*outcome-history+exit*) are higher than that of most of the other predictors with the exception of the simple address-based predictor. For this predictor, the inclusion of the indirect-exit IDs was expected to improve the prediction accuracy of indirect branches due to the additional correlation provided by the indirect-exit IDs along with the regular block exit IDs. However, the reduction in the effective number of hyperblock exits in the global exit history due to the inclusion of indirect-exit IDs seemed to have a negative effect on the accuracy when compared to a regular global exit history based indirect predictor (*outcome-history-based*). For larger indirect predictor sizes, we can use longer history lengths and hence, for 8 KB and 16 KB predictors the *outcome-history+exit* predictor performs almost on par with the *outcome-history-based* predictor. More design-space exploration, especially with larger predictors is required to determine the potential of the *outcome-history-based predictor*.

When considering the overall mean across all the SPEC integer benchmarks, the differences between the various predictors become less prominent. The overall mean of the subset of benchmarks shown in Figure 4.13(b) which have significant numbers of indirect branches and calls (*crafty*, *gcc*, and *perlbmk*) indicate a greater difference in the performance of the different predictors. We chose these three benchmarks based on our analysis results from Chapter 3 shown in Figure 3.32. The path-based predictor is the best for the 256-byte and 512-byte configuration with mean MPKI values of 6.87 and 5.94 respectively. The *EIshift* predictor is the best for the next two sizes (1 KB and 2 KB) with MPKI values of 5.38 and 5.30. For the 4 KB, 8 KB, and 16 KB configurations, the *EIpath* predictor out-

performs the other predictors with MPKIs of 5.21, 4.74, and 4.70 respectively. In the 8 KB category, *EIpath* is 8.3% better than the second-best path history-based predictor (*path-based* with an MPKI of 5.17).

On average, compared the address-based predictor, the best indirect predictors, even for a small budget of 1 KB, can achieve almost 42% reduction in MPKI when considering the subset of three benchmarks. The reduction for 1 KB indirect predictors when considering the entire integer suite is 20.2%. These results show the importance of indirect branch predictors as a component of target predictors. The *EIpath* and *EIshift* techniques are two of the best indirect branch predictors from this set of experiments. They outperform the other predictors when considering predictors that have at least 1 KB of storage. In the next subsection, we compare these two predictors with a state-of-the-art ITTAGE [62] indirect branch predictor. We also propose an exit-inspired predictor that uses the TAGE exit predictor for the indirect-exit prediction.

Comparing exit-inspired predictors with a state-of-the-art indirect branch predictor

In the previous set of experiments, we evaluated several simple indirect branch predictors. We also proposed the exit-inspired indirect branch predictor that can be designed using any exit predictor. We now compare the two best indirect predictors from the previous section (*EIpath* and *EIshift*) with the ITTAGE (Indirect Target TAGE) indirect branch predictor [62].

The ITTAGE predictor is one of the best branch predictors and Seznec et al. show that it can perform better than several previously proposed predictors [62]. The ITTAGE predictor uses the same structure as a TAGE branch predictor, but stores target addresses instead of branch direction bits in the component prediction tables. It can also be easily combined with the TAGE branch predictor so that direction bits and target addresses are maintained in each entry.

Along with the above three predictors, we compare a new exit-inspired predictor called *EITAGE* which uses the TAGE exit predictor described earlier to predict indirect-exits. This is similar to other "EI" predictors where the indirect-exit is predicted first using an indirect-exit predictor and finally, the indirect target is predicted by indexing the indirect BTB with a hash of the block address and the indirect-exit. Just as an *EIpath* predictor uses path history-based exit prediction to predict indirect-exits, the *EITAGE* predictor uses the TAGE-like exit predictor to predict indirect-exits. This is a good point of comparison as the *ITTAGE* predictor and the TAGE exit predictor have a similar structure.

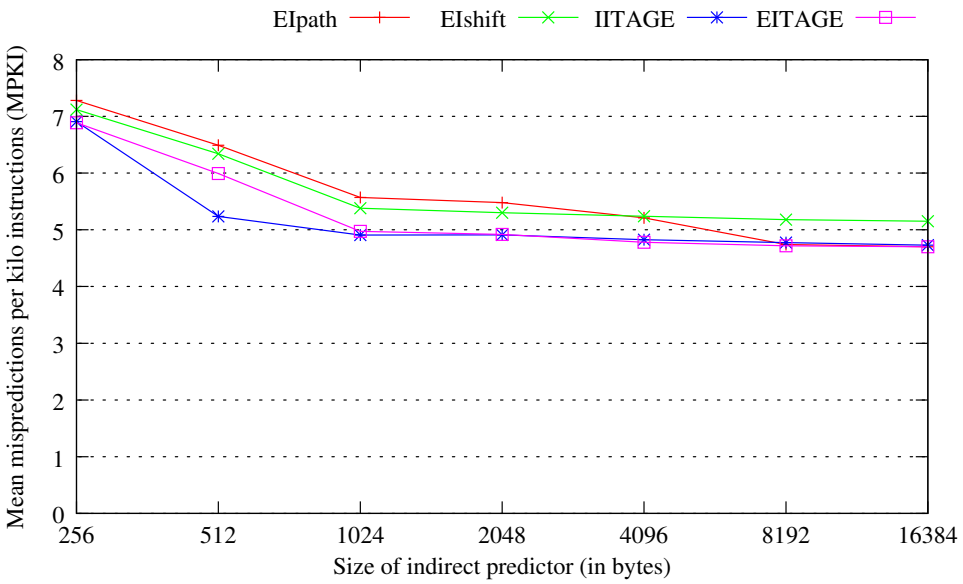


Figure 4.14: Overall MPKI for three different indirect branch predictors for sizes ranging from 256 bytes to 16 KB. Evaluation is done only for the subset of benchmarks (gcc, perlbmk, crafty). The exit predictor is a 16 KB local/global tournament predictor. ITTAGE predictor, Exit-indexed path history-based predictor, and Exit-indexed TAGE predictor are shown.

The above four predictors are compared for various predictor sizes from 256 bytes to 16 KB in Figure 4.14. The comparison is done only for the previously shown subset of integer benchmarks since the other benchmarks do not show a significant percentage of indirect branches or calls. For all the sizes except the 16 KB size, the *EITAGE* uses approximately equal storage for the TAGE indirect-exit predictor and the IBTB table. For

the 16 KB size, it uses only 25% of the storage for the IBTB.

The general trend from the graphs is that for small sizes the *ITTAGE* and *EITAGE* predictors perform better than the rest. For larger sizes, all predictors except *EIshift* perform almost as well as each other. For the 256-byte configuration the best predictors are the *ITTAGE* and the *EITAGE* predictor with an MPKI of 6.9. For the 512-byte and the 1 KB sizes, the *ITTAGE* predictor achieves the lowest MPKI. The two predictors perform similarly for the 2 KB configuration. For higher sizes, the *EITAGE* predictor has the lowest MPKI and slightly outperforms the ITTAGE predictor. Beyond 2 KB, the reduction in MPKI is not significant.

The *EIpath* (and the *EIshift*) predictor has a simple designs consisting of a single indirect-exit prediction table and an indirect BTB in contrast to the *ITTAGE* predictor that has several tagged-tables, different histories and indices and the *EITAGE* predictor that is similar to the *ITTAGE* predictor but has an IBTB buffer to store targets. With a simple design the *EIpath* and *EIshift* predictors are able to achieve MPKIs close to the MPKI of the TAGE-based predictors. For example, in the 1 KB configuration, the *ITTAGE* predictor has an MPKI of 4.91 while the *EIshift* predictor has an MPKI of 5.38 which is 9.6% higher. In the 8 KB configuration, *EIpath* (4.74 MPKI), and *EITAGE* are nearly equal (4.72 MPKI). Hence, if the design constraints require a simple but good design that can achieve the nearly-best MPKI, the *EIpath* or *EIshift* predictors are desirable.

Evaluating indirect predictors with perfect exit prediction

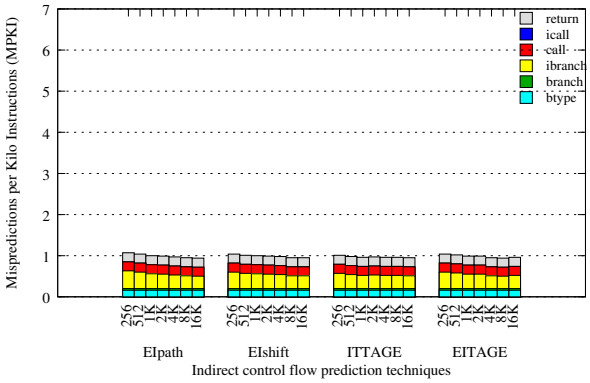
So far we evaluated various indirect branch predictors with the 16 KB local/global tournament exit predictor and the 16 KB improved target predictor with longer offset widths (as in Chapter 3). To isolate the effectiveness of the indirect branch predictor as a component of the target predictor, we simulate the four predictors from the previous set of experiments with a perfect exit predictor. A similar analysis with perfect-exit and realistic-target predictors was done in Chapter 3. We show results for *crafty*, *gcc*, and *perlbmk* in Figure 4.15.

Results for the (subset) mean of these three benchmarks are shown in Figure 4.16.

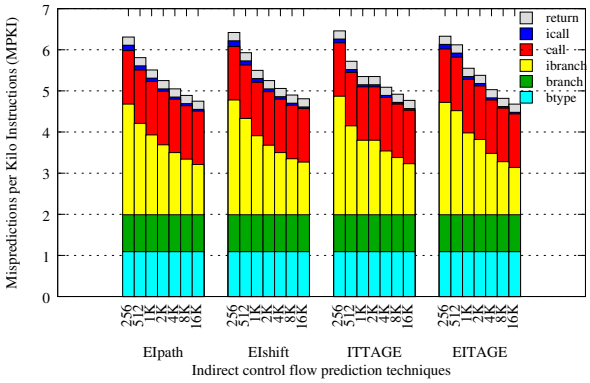
Figure 4.15 shows that for *crafty* there is a slight improvement in target MPKI due to the reduction in indirect branch MPKI as the indirect predictor size is increased. All the four predictors perform almost similarly for *crafty*. With the *EIpath* predictor, the indirect branch MPKI goes down from 0.43 to 0.30 when scaling from 256 bytes to 16 KB resulting in a 12.1% reduction in the overall target MPKI for *crafty*. For *gcc*, we can observe a very high fraction of indirect branch mispredictions and a small fraction of indirect call mispredictions. The indirect branch MPKI is uniformly reduced across all the four predictors when the predictor size is increased. The most effective predictor for small sizes is the *EIpath* predictor while the lowest MPKI at the 16 KB size is from the *EITAGE* predictor. The *EITAGE* predictor achieves a reduction of 58% in the MPKI when the predictor is scaled from 256 bytes to 16 KB. The remaining mispredictions are not removed by any of the predictors. Scaling the predictors, considering alternative designs and more tuning may be necessary to remove the rest of the indirect branch mispredictions. A small percentage of indirect calls is also present in *gcc*. The indirect call MPKI goes down from 0.11 to 0.04 for *EITAGE* when the predictor is scaled from 256 bytes to 16 KB.

Out of the three benchmarks, the maximum impact due to the addition of an indirect branch predictor is seen for *perlbmk* which has indirect calls contributing to almost 100% of the target mispredictions. While all the predictors can remove more than 50% of the indirect call mispredictions for *perlbmk* when considering sizes from 1 KB and beyond, the most effective predictors for *perlbmk* are *ITTAGE* and *EITAGE* as they are capable of removing all the indirect call mispredictions with 1 KB and larger predictors.

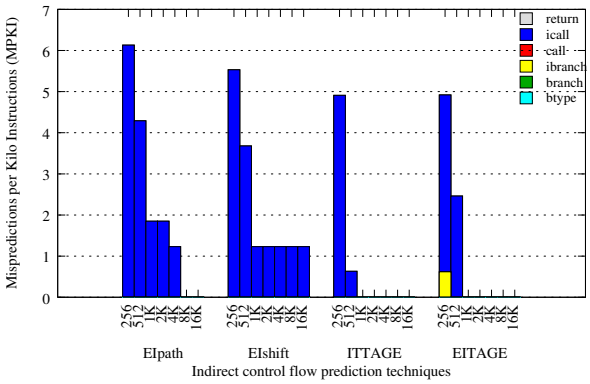
Figure 4.16 shows the mean of the above three benchmarks' MPKIs. On the whole, with perfect exit prediction, a reduction of more than 50% in the target MPKI is observed when moving scaling the predictors from 256 bytes to 16 KB. For example, there is 58.1% reduction in the MPKI when scaling the predictor from 256 bytes to 16 KB for *EIpath*. However, beyond 2 KB or 4 KB, the improvements do not scale well as the indirect predic-



(a) SPEC2K *crafty*



(b) SPEC2K *gcc*



(c) SPEC2K *perlbmk*

Figure 4.15: Target MPKI breakdown for *crafty*, *gcc*, and *perlbmk* for indirect branch predictor sizes ranging from 256 bytes to 16 KB. The exit predictor is perfect while the target predictor is an improved 16 KB target predictor with increased BTB and CTB offset lengths. Four indirect predictors are shown: *EIpath*, *EIshift*, *ITTAGE*, and *EITAGE*. Each stack has the following components from bottom to top: branch type MPKI, regular branch MPKI, indirect branch MPKI, regular call MPKI, indirect call MPKI, and return MPKI.

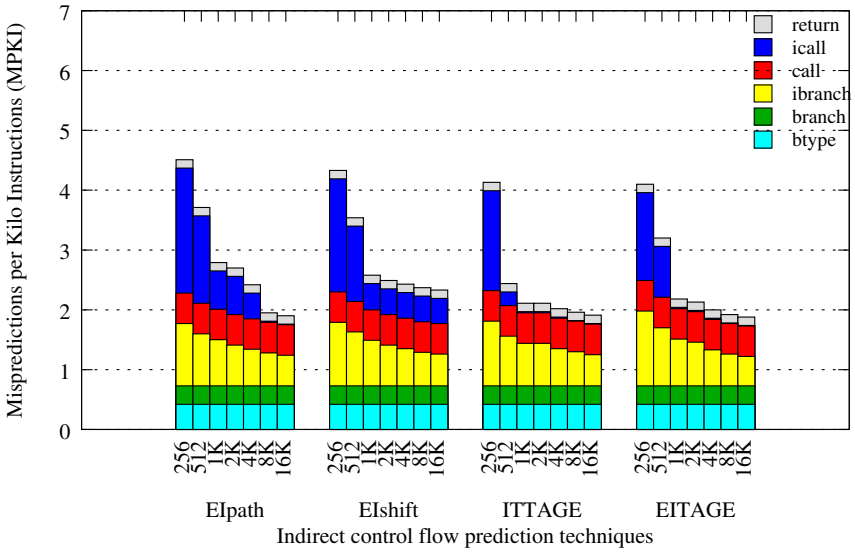


Figure 4.16: Target MPKI breakdown for the subset mean (mean for *crafty*, *gcc*, and *perlbmk*) for indirect branch predictor sizes ranging from 256 bytes to 16 KB. The exit predictor is perfect while the target predictor is an improved 16 KB target predictor with increased BTB and CTB offset lengths. Four indirect predictors are shown: *EIpath*, *EIshift*, *ITTAGE*, and *EITAGE*. Each stack has the following components from bottom to top: branch type MPKI, regular branch MPKI, indirect branch MPKI, regular call MPKI, indirect call MPKI, and return MPKI.

tor is scaled. Hence predictor sizes between 1 KB and 4 KB may be good size/performance trade-off points.

Overall, the results from this section show that significant improvements can be achieved when applying the exit predictor-like concept of predicting an index or an ID first followed by predicting the target address. While saving space, this two-stage design also offers an amenable way to optimize the exit prediction stage using complex and hybrid predictors in the first stage and leaving the target prediction to a simpler BTB-like structure directly indexed using the results of the previous stage. This technique is not only applicable to indirect branch prediction but is also extensible to other predictions like load value prediction and return value prediction.

4.6 Combining block predictor component improvements

In the previous sections, we have discussed several improvements for the exit predictor and the target predictor. In Chapter 3, we showed some that by adding some bits to the offset entries in target buffers, we can achieve significant reduction in the MPKI. In this section, we combine all these component improvements discussed in this chapter and in Chapter 3 to arrive at an improved 32 KB block predictor. Enhancements for the 10 KB prototype predictor are presented in [49].

We choose the 32 KB scaled-up prototype predictor (called *proto32K*) presented in Chapter 3 as our baseline predictor. This predictor achieves a mean exit MPKI of 4.48 and an overall MPKI of 6.49 for SPEC integer benchmarks, and an exit MPKI of 1.2 and an overall MPKI of 1.3 for SPEC FP benchmarks. The *proto32K* predictor was a direct scaling-up of the 10 KB TRIPS prototype predictor. We identified some simple techniques with which we can reduce the MPKI of the *proto32K* predictor like increasing offset widths. We combine such simple improvements with better prediction techniques for different components while incurring little storage overhead. Our improved predictor is called *imprv32K*. This predictor has the following components and differences when compared to the *proto32K* predictor:

- For the exit predictor component, we choose an asymmetric PPE exit predictor described earlier. The PPE predictor uses piecewise-linear components as described earlier in this chapter and is slightly over 16 KB in size. This predictor is similar in structure to the 16 KB PPE described earlier in this chapter but slightly smaller. The earlier 16 KB PPE predictor in this chapter used almost 18 KB of storage (since a 15% overhead was allowed).

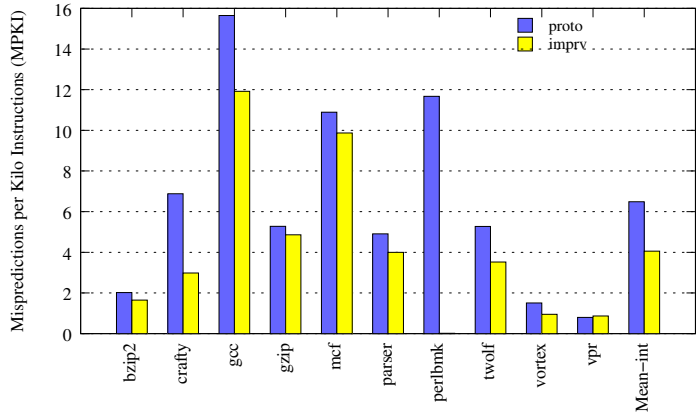
- The branch type predictor includes one extra bit in each entry to be able to predict indirect branches and indirect calls also. In *proto32K*, the branch type predictor could predict only four types. In *imprv32K* it predicts six types. The branch type predictor

also uses only half the number of entries as in *proto32K* because we want the table storage overhead to be as small as possible when compared to *proto32K*. We found that that reducing the number of entries only affects the MPKI by less than 1%.

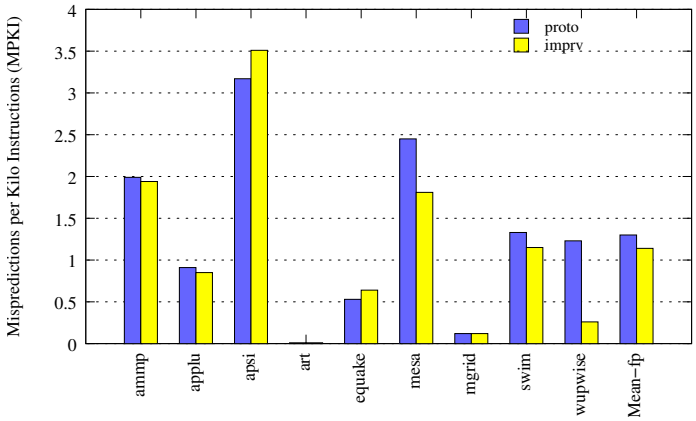
- The sequential predictor remains the same for *imprv32K*.
- The BTB has the same number of entries as *proto32K* but includes longer offsets. The branch offset size in each entry is increased from nine bits to 13 bits as discussed in Chapter 3.
- The CTB has the same number of entries as *proto32K* but includes has one extra bit for the return address offset in each entry (as in Chapter 3).
- The RAS has extra bits in each entry to support the increased return offset length. The number of entries in the RAS is cut down by half to provide for space for the increase in the storage for other components.
- An indirect branch predictor of size 2 KB is added. For this, we pick one of our best exit-inspired predictors, the *EITAGE*. This predictor can reduce almost as many mispredictions with a 2 KB configuration as it can in a 16 KB configuration.

Figure 4.17 compares the overall MPKI of *proto32K* and *imprv32K* for SPEC integer and FP benchmarks. For the integer suite, except for a slight increase in the MPKI for *vpr*, there is significant reduction in MPKI for all the benchmarks. On the average, the overall MPKI goes down from 6.49 for *proto32K* to 4.06 for *imprv32K* which represents an improvement of 37.4%. For the FP suite, other than *apsi*, all the benchmarks have slight reduction in their MPKIs. On the average, the FP MPKI goes down from 1.30 for *proto32K* to 1.14 for *imprv32K*, an improvement of 13.8%.

To look at the reasons for the MPKI reductions in more detail, we compare the MPKI breakdown for both the predictors in Figure 4.18 for SPEC integer and floating-point benchmarks. As stated in Chapter 3, the breakdown graph isolates the MPKI of different



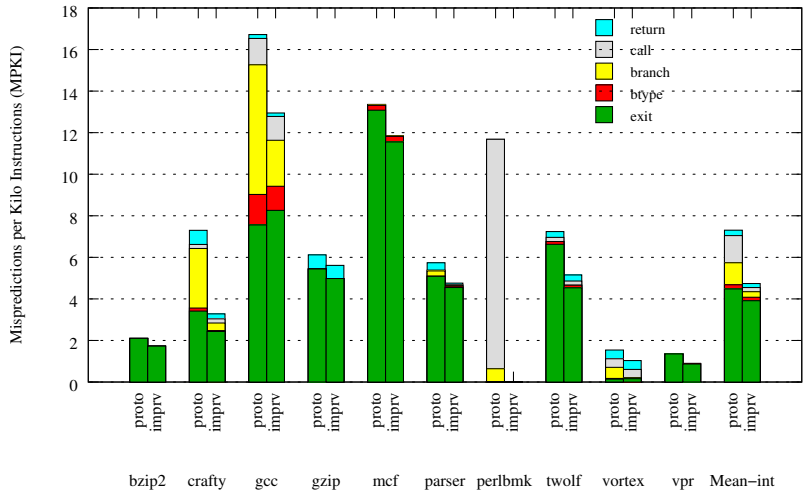
(a) SPEC2K integer benchmarks



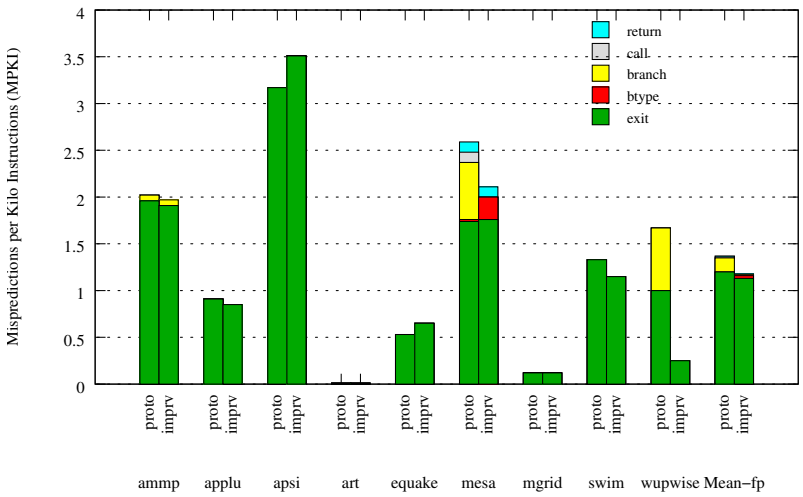
(b) SPEC2K FP benchmarks

Figure 4.17: Comparison of overall MPKI of the scaled-up prototype block predictor (*proto32K*) with the improved block predictor (*imprv32K*).

components so that bottlenecks in various components can be understood. The sum of the individual component MPKIs is larger than overall MPKI in Figure 4.17 because the exit MPKI derived from the exit predictor component is represented exactly in the graph. The overall predictor MPKI is lower because some exit mispredictions are hidden by the target predictor in situations when the target prediction is right even if the exit prediction is wrong. The purpose of this breakdown comparison is to see the relative improvements in each component leading to the overall improvement. The stack for each benchmark has five components representing (from bottom) the exit MPKI, branch type MPKI, branch



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 4.18: Comparison of MPKI breakdown by component for the scaled-up prototype block predictor (*proto32K*) and the improved block predictor (*imprv32K*). The five components, from the bottom, in each stacked bar represent t the exit MPKI, branch type MPKI, branch MPKI (regular and indirect branch), call MPKI (regular and indirect call), and return MPKI. Results are shown for SPEC integer and FP benchmarks.

MPKI (includes regular branch and indirect branch MPKI), call MPKI (includes regular call and indirect call MPKI), and return MPKI.

For both sets of benchmarks, the majority of the mispredictions (82.5% for the integer suite and 95.8% for the FP suite) are from the exit predictor as seen in the misprediction breakdown graph. However, several of the exit mispredictions are hidden by correct target predictions. Taking this effect into account, about 66.5% and 95.6% of the overall mispredictions (for integer and FP benchmarks respectively) are directly because of exit mispredictions. When compared to *proto32K*, there is a 12.7% reduction in the mean exit MPKI for the integer suite and a 5.8% reduction for the FP suite for the *imprv32K* predictor. There is a significant increase in the exit MPKI for *gcc* but the overall MPKI is lower due to the indirect branch MPKI improvements. There are several benchmarks which have other components contributing to a significant fraction of the mispredictions. For example, for *gcc* and *mcf*, branch type mispredictions are present.

The integer benchmark *crafty* has a high branch MPKI (because of indirect branches) which is reduced to a small fraction in the *imprv32K* predictor. For *perlbmk*, all the mispredictions due to branches and calls are removed. As we saw in the previous section, most of these mispredictions are due to indirect calls. The branch mispredictions in *vortex*, *mesa*, and *wupwise* are mainly due to the BTB offset length mispredictions which are all removed in the *imprv32K* predictor. The return mispredictions component is reduced for crafty due to the increase in the return offset in the CTB. For benchmarks like *gcc*, *gzip*, *twolf*, *vortex*, and *mesa*, RAS mispredictions are seen in *imprv32K*. These mispredictions may be due to first-time return address misses (compulsory misses before the return addresses are learned in the CTB) or stack overflow/underflow.

Though the above results are promising, there is still a long way to go to eliminate the majority of the mispredictions. Most of the remaining mispredictions are from the exit predictor. More exit predictor exploration with existing and new techniques may be required to reduce the exit MPKI. Another way to reduce the number of mispredictions is

to increase the predictability of hyperblock exits, which is discussed in the next chapter.

4.7 Summary

In this chapter we used the insights from our analysis in Chapter 3 and proposed improved exit and target prediction techniques. We evaluated previously proposed and new chooser predictors which have the capability to provide lower misprediction rates than traditional global history-based choosers. The best chooser applied to a hybrid-4 multi-component predictor offered 4% reduction in MPKI for the integer suite. Next, we presented techniques to adapt state-of-the-art branch predictors to exit prediction. We showed that some techniques, like the GEHL-like prediction scheme, offer hardly any improvement over the local/global tournament exit predictor due to inefficiencies when mapping to exit prediction. Better improvements may be possible if efficient choosers, alternative OGEHL configurations, and larger sizes are explored. Other techniques like the TAGE-like predictor and the 8-perceptron predictor were more promising. A local/TAGE tournament exit predictor offered about 5% reduction in exit MPKI for a 16 KB size and almost 13% reduction in exit MPKI for a 64 KB size, compared to an equal-sized local/global tournament predictor. The improvement offered by the 8-perceptron predictor was higher for the 64 KB size.

To directly use branch direction predictors (binary predictors) instead of adapting them to exit prediction, we proposed using a Post Prediction Encoding technique that uses branch predictors to predict each bit of an exit. The final predicted exit (encoded exit ID) is the concatenation of the predictions from the three component predictions. This technique combined with the piecewise-linear branch predictor as its component predictor offered the lowest MPKI (8% for the integer suite and 6.8% for the floating-point suite) for the 16 KB size.

In the last chapter, we had identified indirect branch/call mispredictions as the primary source of the remaining target mispredictions. In this chapter we evaluated various types of indirect branch predictors using global and path histories. We evaluated several

predictors from the literature such as global history-based two-level predictors, path-based two-level predictors, and the ITTAGE indirect branch predictor. Inspired by exit prediction followed by target prediction for making block predictions (instead of directly predicting the target address), we proposed a two-stage scheme for indirect branch predictors. In the first stage, using any of the exit predictors, an indirect-exit ID is predicted. Using this ID and the block address, the indirect target address is predicted in the second stage. The exit-inspired indirect branch predictors performed as well as or better than predictors of comparable sizes.

Finally, we showed a comparison of the scaled-up 16 KB prototype block predictor with a 16 KB improved predictor consisting of the PPE exit predictor, longer offsets, and a 2 KB indirect branch predictor. This predictor achieved 37.4% reduction in MPKI for the integer suite and 13.8% for the FP suite when compared to the scaled-up prototype predictor. Misprediction breakdown for the improved predictor showed that most of the remaining mispredictions are due to exit predictor inefficiency due to non-optimal exit prediction technique or design and exit predictability issues. About 66.5% of the mispredictions for integer benchmarks are because of exit mispredictions. We explore exit predictability using correlation analysis in the next chapter.

Chapter 5

Analysis of Correlation in Hyperblocks

In the last few chapters, we performed a systematic analysis of exit and target predictors and proposed hardware prediction techniques to improve the performance of block predictors. Our results showed that, by carefully designing block predictors, lower misprediction rates can be obtained compared to the misprediction rates from the TRIPS prototype predictor. However, the misprediction rates from the best block predictors described in Chapter 4 are still significantly higher than the typical misprediction rates for the best conventional branch predictors [13, 24, 60, 62]. Some of the best exit predictors we evaluated had designs similar to state-of-the-art conventional branch predictors but they could not match the performance reported in previous work. In this chapter we attempt to understand whether using history-based predictors to get accurate predictions is inherently more difficult for exit prediction (for hyperblocks) than branch prediction (for basic blocks). We examine the role of hyperblock formation in poor block predictability. We perform branch correlation analysis and propose software and hardware approaches to improve exit predictability.

History-based predictors learn the behavior of past branches and previous executions of the current branch. The collected information in the history registers are used to

predict the current branch. This technique works well because most branches in a program are either correlated with previous executions of the same branch (self-correlation) or previous execution of other branches (cross-correlation). Local branch and exit predictors exploit self-correlation while global branch and exit predictors exploit cross-correlation. While a global branch/exit predictor explicitly uses global correlation information found in global histories, a path-based branch/exit predictor uses information about the path leading to the current branch/block to make a prediction. Hence, a path-based predictor exploits the correlation of the global path through the program leading up to a branch/block to make a prediction for that branch/block.

We use perceptron-based correlation analysis to show that the loss of global correlation due to correlation-agnostic hyperblock construction is a key cause for the relatively poor performance of exit predictors. Our study uses perceptrons to track correlation among branches and capture the distribution of correlated branches. We also point out the importance of local predictors in capturing intra-block correlation. We suggest hardware and software techniques that can be used to improve predictability using correlation-aware block formation and prediction.

In this chapter, we limit our study to integer benchmarks, as we find very little difference between the performance of exit and branch predictors for floating-point programs. We analyze only exit predictability in this chapter. Loss of correlation may impact some forms of target prediction, for example, global history-based indirect branch prediction. However, we do not consider target prediction in our analysis.

5.1 Comparison of exit and branch predictability

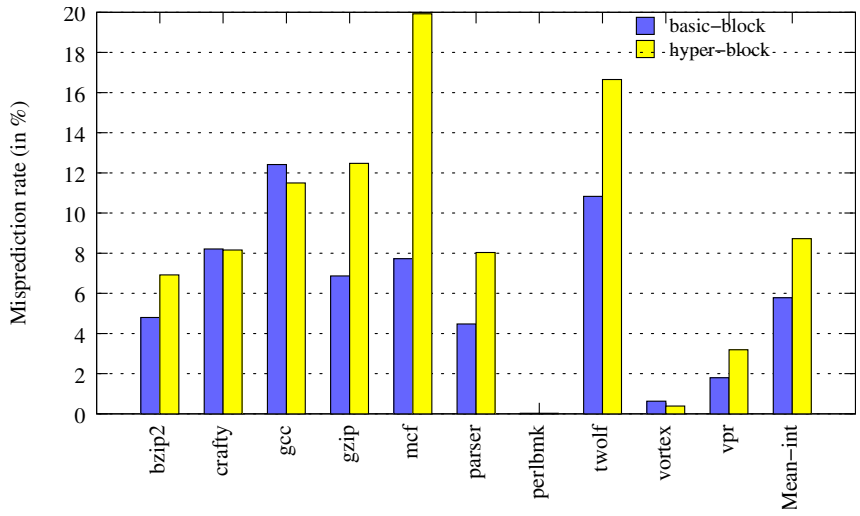
One of the goals of hyperblock formation is to improve predictability by hiding difficult-to-predict branches as predicates, thereby improving the performance of the processor. It is important to note that improving the predictability of blocks alone does not guarantee increased performance. A trivial solution to improve prediction accuracy would be to use

basic blocks as the unit of execution. Since branch predictors can in general offer better accuracy, this approach would be good for predictability. The downside is that basic blocks are very small and offer very little scope for aggressive optimizations. Further, when using basic blocks, the instruction window is severely under-utilized in a block-based processor like TRIPS, leading to degradation in performance. On the other hand, combining too many basic blocks to construct big hyperblocks using predication [1] might create several exit branches which might lead to poor predictability. Hence a delicate balance has to be struck between the number of useful instructions in a block and block predictability.

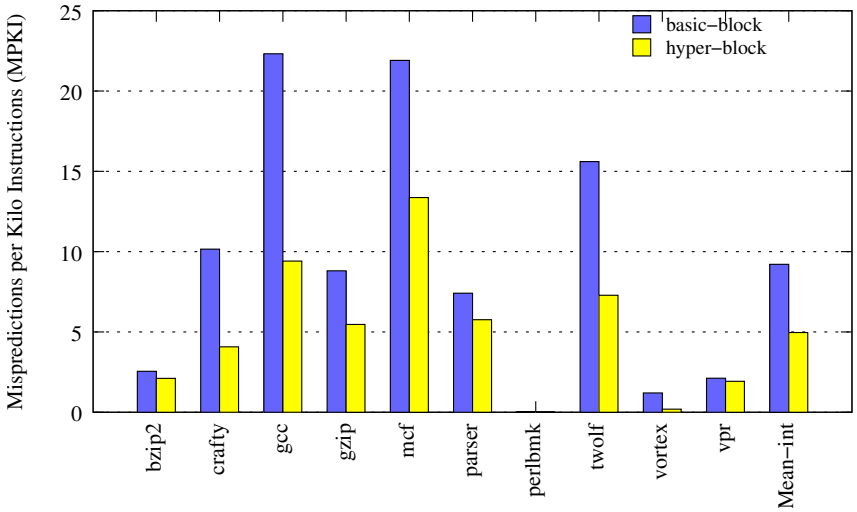
Predicting exits can be more difficult than predicting branches because of possible loss in correlation among exits due to predication. Further, predicting exits is a "$1-of-N$" problem (where N is 8 for the TRIPS prototype) while predicting branches is a "$1-of-2$" problem. We evaluate prediction accuracy for TRIPS basic block (BB) code (compiled with -O3 option in Scale [68]) and predicated hyperblock (HB) code (compiled with -Omax). Both the codes are evaluated for the same simpoint simulation region as discussed in Chapter 2.

We compare misprediction rates as well as total mispredictions for exits and branches respectively, from the HB and BB compilations. For each comparison, we use similar predictors of the same size for exits and blocks. The number of entries in the branch predictor prediction tables (second level) would be double the number of entries in the exit prediction tables as each entry is only two bits wide (for two-bit saturating counters) compared to four bits (three bits for exit and one bit for hysteresis). This comparison is fair as we use the same style of predictors of the same sizes for both basic blocks and hyperblocks.

To motivate our BB and HB code predictability comparison, we first show exit misprediction rates and exit MPKI from the TRIPS prototype predictor. The exit predictor component in the TRIPS prototype predictor is 5 KB in size. We construct a 5 KB local/global tournament branch predictor similar in structure to the prototype exit predictor. The misprediction rate and MPKI for the 5 KB exit and branch predictors are shown in Fig-



(a) Exit misprediction rates



(b) Exit MPKI

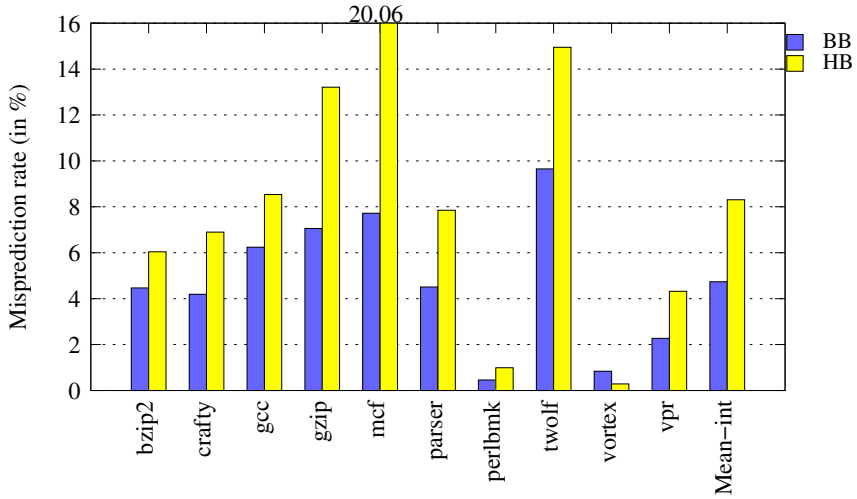
Figure 5.1: Comparison of exit misprediction rates and exit MPKI for the 5 KB TRIPS prototype exit predictor for hyperblocks and a 5 KB local/global tournament branch predictor for basic blocks for SPEC integer benchmarks

ure 5.1. The mean exit misprediction rate (Figure 5.1(a)) for the HB code is 7.97% while the mean misprediction rate for the BB code is 4.79%. The tournament branch predictor of the same size performs 40% better than the tournament exit predictor. The MPKI values (Figure 5.1(b) indicate an opposite trend. The mean MPKI for HB code is 4.48 while it is 7.68 for the BB code. The exit predictor MPKI is 42% lower compared to the branch predictor MPKI. This is because, even though the exit predictor is predicting worse than the branch predictor, it has to make many fewer predictions which leads to a large reduction in the MPKI.

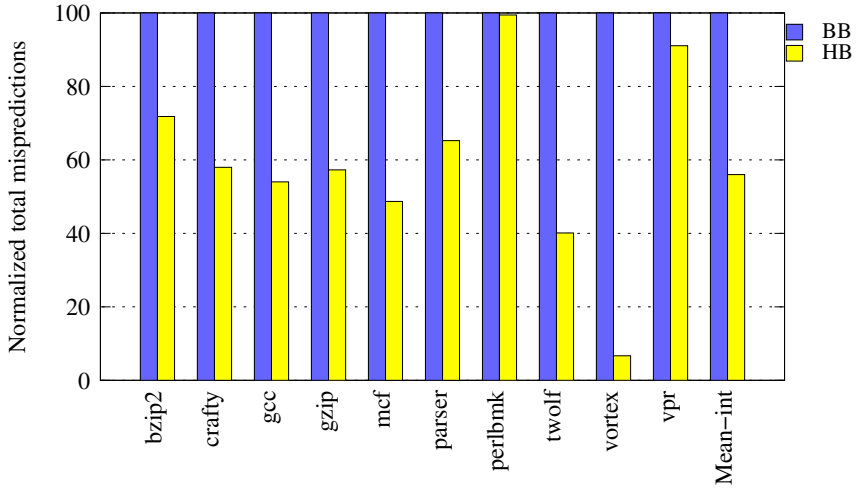
We now compare realistic (16 KB) and interference-free local and global branch/exit predictors.

Global correlation

Figures 5.2 and 5.3 show the misprediction rates and relative misses for the realistic and interference-free global predictors. The exit predictors use a 15-bit global exit history while the branch predictors use a 16-bit global branch history. The absolute misprediction rates are plotted for basic block (BB) and hyperblock (HB) code. We also show the total number of misses normalized to the number of misses in the basic block code. The misprediction rates from Figure 5.2 for HB code and BB code are 8.31% and 4.74%. The misprediction rate for the BB code is 43% lower with a similar sized predictor. To remove aliasing effects, we simulated interference-free predictors. When there is no aliasing, the misprediction rates (from Figure 5.3) are 7.37% and 4.0% for HB and BB code respectively with the BB misprediction rate being 45.7% lower. The big difference shows that the global exit predictor is somehow not able to exploit correlation as effectively as the global branch predictor. One way to exploit better correlation is by increasing the history length. Even if the history length of the interference-free global exit predictor is doubled to 30 bits, we only obtain a misprediction rate of 6.29% which is still significantly higher than the misprediction rate from the global branch predictor using a 15-bit history. Lower misprediction rates

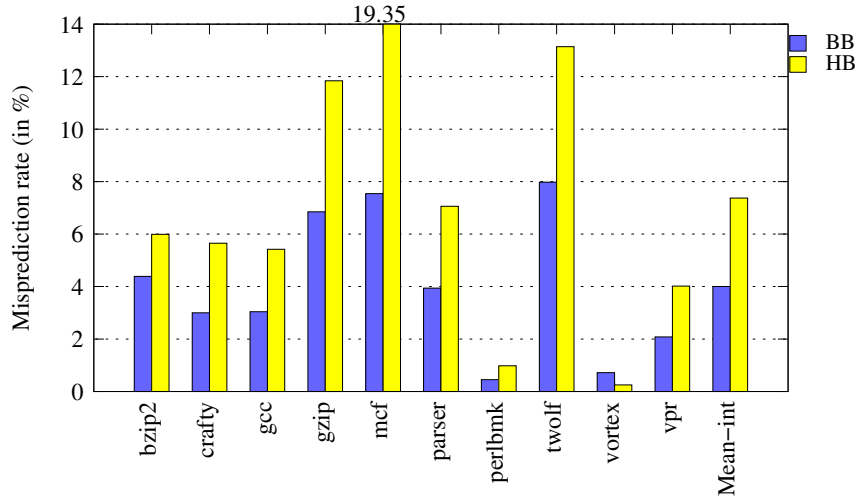


(a) Exit misprediction rates

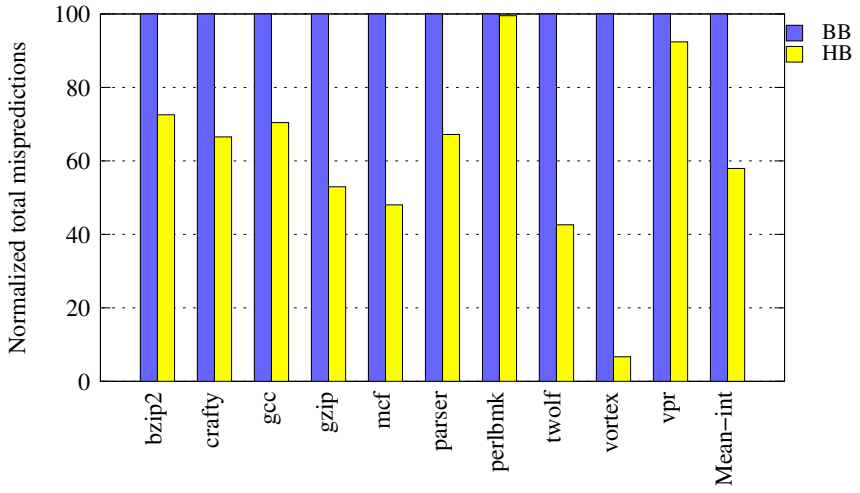


(b) Total exit mispredictions

Figure 5.2: Comparison of misprediction rates and total normalized mispredictions for 16 KB global exit predictor for hyperblocks and 16 KB global branch predictor for basic blocks



(a) Exit misprediction rates



(b) Total exit mispredictions

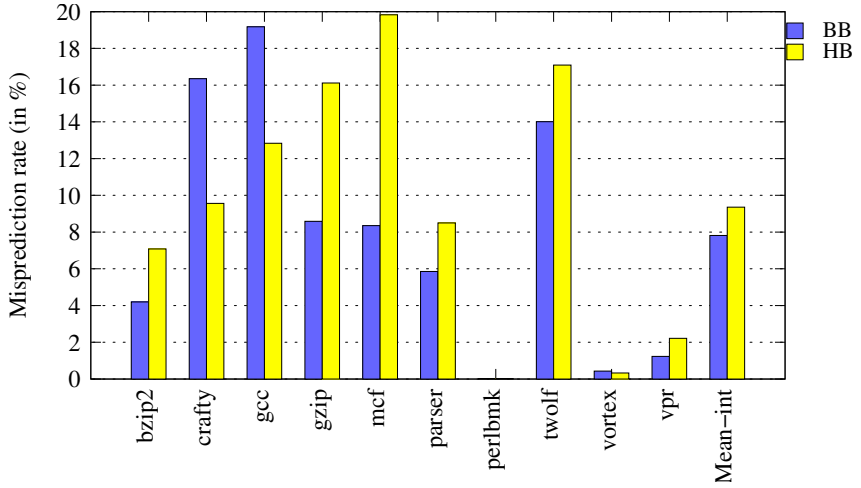
Figure 5.3: Comparison of misprediction rates and total normalized mispredictions for interference-free global exit predictor (with 15-bit history) for hyperblocks and interference-free global branch predictor (with 16-bit history) for basic blocks

indicate that the branch predictor is inherently better at making predictions using global histories.

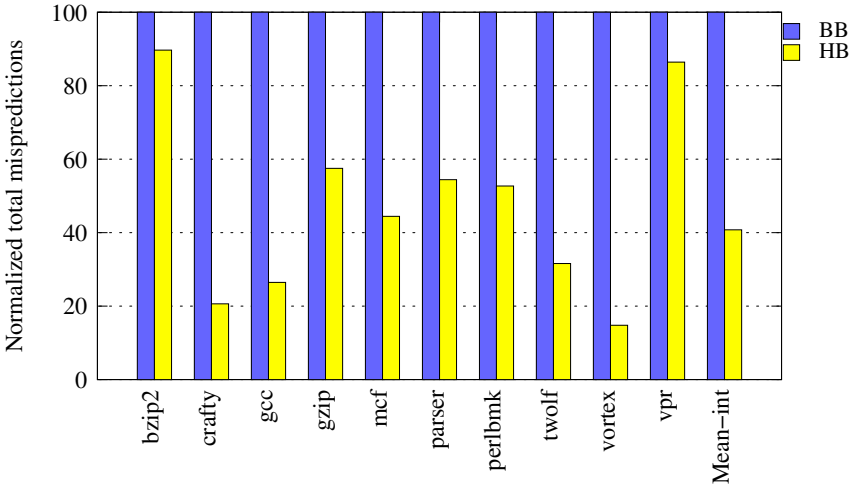
On the other hand, the total number of mispredictions in the simulated code indicate an opposite trend. Figures 5.2 and 5.3 show that the number of misses in the HB code is 44% lower with the realistic global predictor and 42% lower with the interference-free predictor. The total number of mispredictions is much lower in the hyperblock code as it makes far fewer predictions compared to basic blocks. The average number of predictions in the HB code is 67% lower than the number of predictions in the BB code. Hence, even though the exit predictor is not making predictions as effectively as the branch predictor, it is making less than a third of the number of basic block predictions which leads to a reduction in the total number of misses. Since the total number of misses (or MPKI) is the real indicator of the impact of branch mispredictions on performance, we see that the hyperblock code is, in effect, better performing than the basic block code. It also justifies the use of hyperblocks instead of basic blocks in TRIPS for improved performance. If the predictability of exits can be improved to equal or exceed the predictability of basic blocks, we can achieve further reduction in the MPKI, leading to larger performance improvements.

Local correlation

Figures 5.4 and 5.5 show the results of comparison for local exit and local branch predictors. A local exit predictor tracks correlation information local to the hyperblock, i.e., information about previously encountered exits in the hyperblock. A local branch predictor tracks correlation information from the past executions of the current branch. The misprediction rates of the realistic 16 KB local predictor are 9.36% and 7.27%, the branch predictor having 16.5% lower misprediction rate. This difference is not as pronounced as the difference in the global predictors. Since there are far more basic blocks in the BB code than there are hyperblocks in the HB code in a program than there are hyperblocks, we expect to see more aliasing in the local branch predictors than the local exit predictors. This hypothesis is

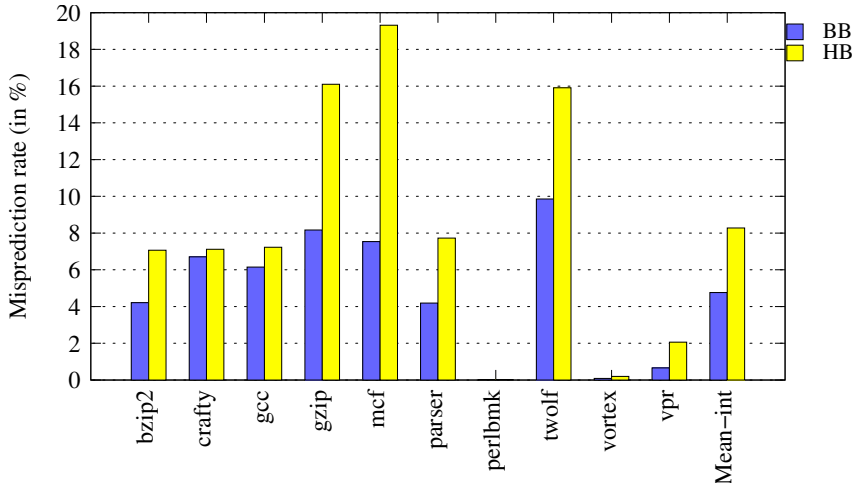


(a) Exit misprediction rates

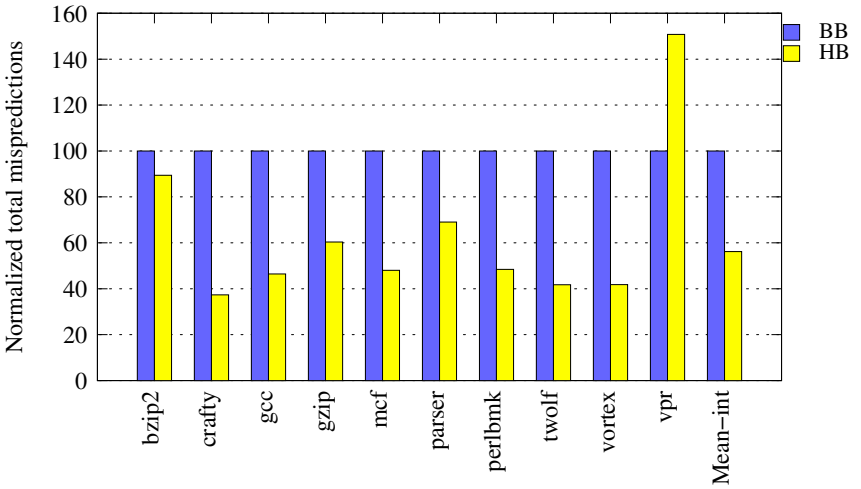


(b) Total exit mispredictions

Figure 5.4: Comparison of misprediction rates and total normalized mispredictions for 16 KB local exit predictor for hyperblocks and 16 KB local branch predictor for basic blocks



(a) Exit misprediction rates



(b) Total exit mispredictions

Figure 5.5: Comparison of misprediction rates and total normalized mispredictions for interference-free local exit predictor (with 15-bit history) for hyperblocks and interference-free local branch predictor (with 16-bit history) for basic blocks

confirmed when seeing the results of local prediction using interference-free tables (history as well as exit prediction tables). The misprediction rates for interference-free predictors are 8.28% and 4.76% respectively, for the exit and branch predictors. In this case, the branch predictor is 42.5% better. As before, the total number of mispredictions indicates an opposite trend: the local block predictor has 59.2% fewer mispredictions with a realistic predictor and 42.8% fewer mispredictions with an interference-free predictor.

Comparing local and global history-based predictors

We now compare the effectiveness of local/global tournament predictors (using ideal choosers) between BB and HB code. We make the chooser ideal as we would like to explore how the local/global combination performs for basic blocks and hyperblocks. We choose the local/global combination as they exploit different types of correlation (self- and cross- correlation). We simulated a combination of interference-free global and local exit and branch predictors described above and shown in Figures 5.3 and 5.5.

The misprediction breakdown comparison is shown in Figure 5.6. The lowermost component of each bar shows the percentage of predictions made incorrectly by both the local and the global components. Both components fail to predict correctly for 1.81% of the dynamic blocks in BB code. However, it is much higher for the HB code, 4.5%. The lowermost components represent the misprediction rate achieved by the local/global combination predictor containing an ideal chooser. These numbers show the inherent inability of both predictors to exploit branch correlation in hyperblocks. The next component (second from bottom) in the bars shows the percentage of blocks for which only the local predictor can predict correctly. The local branch predictor correctly predicts 2.19% of the branches that cannot be predicted by the global predictor. However, the local exit predictor does better: it can predict 2.89% of the branches correctly. Since local exit predictors exploit some global correlation also (as they capture the behavior of the other exits in the same hyperblock as well), they may perform better than local branch predictors. The next component in each

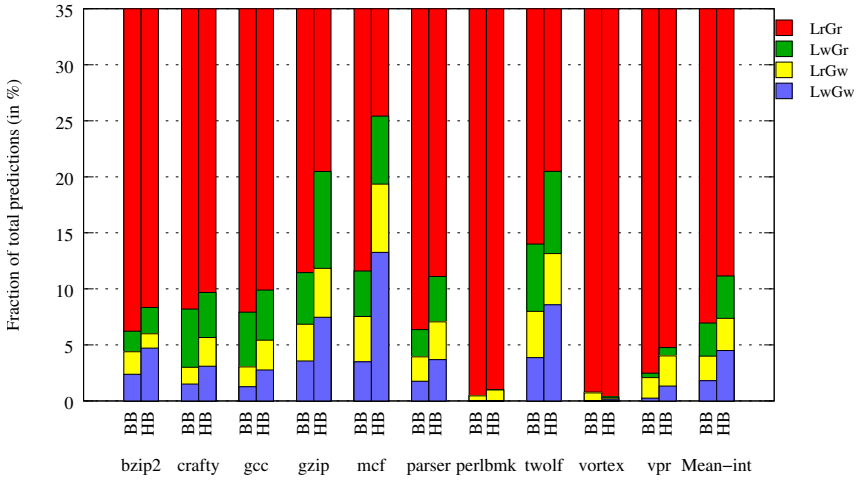


Figure 5.6: Misprediction breakdown comparison between BB and HB code for a local/global tournament predictor with ideal chooser. Both predictors are interference free and use 15-bit histories to predict exits and 16-bit histories to predict branches. The four components in each bar (starting from bottom) indicate the fraction of blocks for which both predictors predict incorrectly, the fraction for which only the local predictor predicts correctly, the fraction for which only global predictor predicts correctly and the fraction for which both predictors predict correctly. The misprediction rate of this local/global tournament predictor with an ideal chooser is given by the lower most component in each bar when both components fail to predict correctly.

bar represents the percentage of time the global predictors can predict correctly while the local predictors are wrong. The global branch predictor is able to predict 2.95% of the branches correctly while the global exit predictor can predict 3.78% of the branches correctly. The last component shows the percentage of branches that both the local and the global predictor can predict correctly. This number is 93.05% for the branch predictor and 88.85% for the exit predictor. This breakdown shows that while the local and global branch predictors can predict majority of the branches correctly, the improvement obtained by using both components instead of say, a single global predictor, is much less compared to the improvement of a local/global tournament exit predictor over a global exit predictor. Hence the presence of a local component may be more important for exit predictors than branch predictors.

5.2 Understanding exit predictability using correlation analysis

We now perform a perceptron-based correlation analysis to track global branch correlation in basic blocks and hyperblocks. To identify the loss of correlation due to block construction mechanisms, we perform a dynamic correlation analysis of each branch in the BB code and identify all the other static branches that strongly influence the direction of this branch. A global perceptron predictor [25] is used to determine the set of correlated branches (in near and far histories) for each branch in the BB code. The compiler tags each branch with a unique ID in the basic block phase of compilation and carries over the tag in all the subsequent phases (including hyperblock formation) until intermediate TIL [68] code generation. Thus, using these tags in the intermediate code from the HB phase, we can determine how the BB branches are mapped to the HB code. The perceptron-based analyzer and the BB tags are used in the following steps for our correlation analysis.

Step 1: Finding correlated branches in BB code

We first run our perceptron-based analyzer on the BB code and track the global correlations for each branch (or basic block). If the direction of the current branch is same as the last seen direction of another branch X, we increment its corresponding perceptron weight. Otherwise, we decrement the perceptron weight. For a given branch, if the weight of a perceptron corresponding to another branch in the history is a large positive number, it indicates a strong positive correlation between the two branches; if the correlated branch is taken, the current branch will also be taken and if it is not-taken the current branch is likely to be not taken. If the weight of the perceptron corresponding to another branch in the history is a large negative number, it indicates a strong negative correlation between the two branches; the direction of the current branch is likely to be opposite to the direction of the correlated branch. h. For each dynamic branch, we initially performed correlation analysis over all the static branches. We found that there are several correlated branches very far away in the history, not typically captured with global-history based predictors

using history lengths less than or equal to 64 bits. Since, typically branch predictors use fewer than 64 bit histories for all or most of the prediction components in the predictor, we modified this strategy to account for correlation only in recently seen branches. Hence for a given branch, we track the correlation using perceptron weights for branches falling within the past 64, 32, and 16 basic blocks seen globally. We choose three history lengths to show the amount of correlation that can be exploited by predictors using short history lengths (up to 16 bits of history) as well as the correlation that can exploited by predictors using long history lengths [24, 60]. When the simulation completes, we identify the set of highly correlated branches for each static branch within the three different history lengths. We use a threshold of 75% to identify branches that are highly correlated with a given branch. A threshold of 75% indicates that the absolute perceptron weight must be at least 0.75 times the dynamic execution weight of the current branch. Our threshold is set to 75% to ensure that branches that are strongly correlated are included in the set. For example, a threshold of 50% indicates that the pair of branches are correlated about half the time. Not choosing a very high threshold above 90% also ensures that we do not include only very strongly correlated branches. We believe that a threshold of 75% captures most of the strongly correlated branches without being too restrictive.

Step 2: Mapping branch addresses to BB tags in BB code

The addresses of correlated branches for each static branch, found in the previous simulation step, are mapped to the BB tags marked by the compiler in the intermediate code for the basic block compilation using the *objdump* of the program binary.

Step 3: Using BB tags for identifying the placement of correlated branches

Since the compiler maintains BB tags until intermediate code generation, we can use the same tags in the intermediate code from the HB compilation phase to determine how the branches from the BB code get mapped to the control flow instructions from the HB code.

Since hyperblocks may have predicated instructions, the branches in the BB code may be present as exit branches or as predicate defining instructions in the HB code. Some branches may be present as both exit branches and predicate defines due to duplication of code (such as tail duplication) while forming hyperblocks [38]. Our tool marks all the correlated tags and the type of the control instruction (branch/predicate-define) for each static branch tag in the BB code.

Step 4: Obtaining weighted placement from HB simulation run

Subsequently, during a simulation run of the HB code, we determine, for each hyperblock exit taken, whether the branches on which it is correlated occurred in the same hyperblock or in a different hyperblock. Further we determine whether each correlated branch occurs as an exit branch or as a predicate-define instruction. We do this for every dynamic hyperblock exit taken and this gives a weighted sum of where correlated branches are placed in the HB code.

We plot the dynamic distribution of correlated branches in HB code execution in Figure 5.7. The figure shows nine of the ten SPEC integer benchmarks used in our studies. We do not show *perlbmk* as it did not work with our correlation-finding infrastructure. As explained before, the perceptron-based correlations are tracked for 16-bit, 32-bit, and 64-bit histories. From this graph we can understand how much correlation information hyperblock codes stand to lose in comparison to basic blocks when using global history-based predictors capturing the last 16, 32 or 64 branches. For a given exit branch X in hyperblock H, a strongly influencing branch can be located in one of the following four places: within the same hyperblock H as an exit branch Y (bottommost component in each bar), within the same hyperblock H as a predicate generating instruction (second component from bottom), within a different block as an exit branch (third component) and within a different hyperblock as a predicate-define instruction (topmost component). The figure shows that except for some benchmarks like *bzip2*, *crafty*, and *vortex*, there is not much

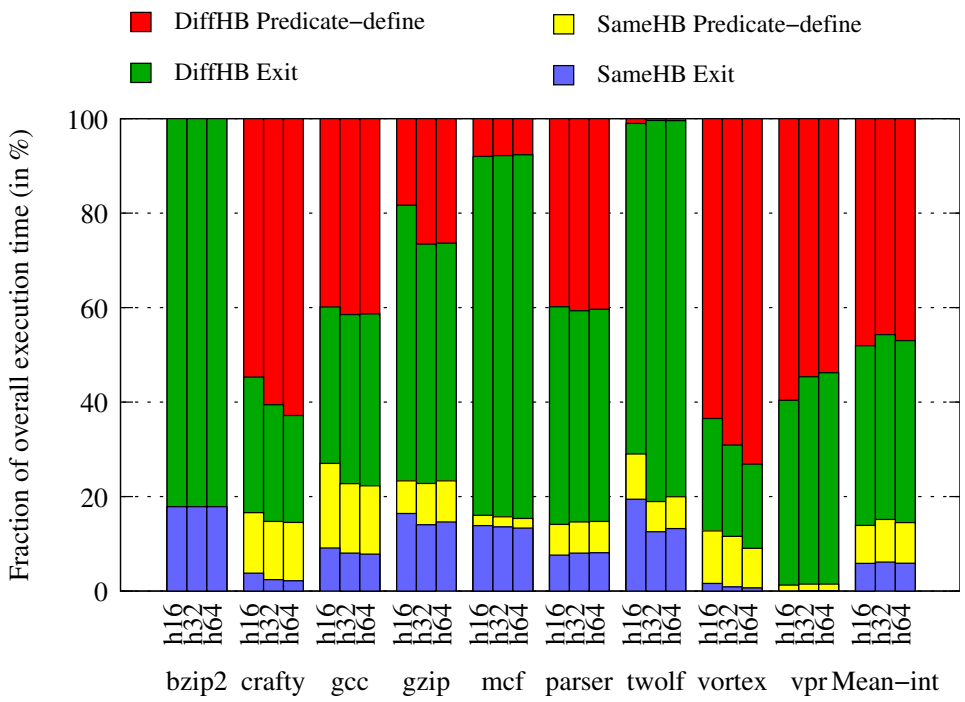


Figure 5.7: Dynamic distribution of correlated branches among hyperblocks. The weighted sum (for each dynamic exit) is normalized and shown.

difference between various history lengths in capturing global correlation. If the correlated branch is in the same hyperblock as the current exit branch, the global predictor may not be able to use this per-block-correlation information of the correlated branch if it did not occur in the recent history captured by the global predictor. If the correlated branch is in the same block or in a different block, but has been converted to a predicate-define, we once again lose correlation information that was present in the BB compiled code. If it does not, a local predictor may prove to be more useful in tracking this correlation.

On average, dynamic correlated branches tend to occur within the same hyperblock as much as 15% of the time as exits or predicate-defines. Note that we do not consider self-correlation in this study. Hence this 20% indicates the influence of other exits or predicate-defines within the same block. The percentage of correlated branches present as predicate-defines within the same block is 6%, while the 9% of the branches are present as exits within

the same block. Some of the correlation provided by exits within the same hyperblock may be captured by a global predictor if the recent history in the predictor captures multiple instances of the current block. However, in general, a local predictor can capture self-correlation more easily.

About 85% of the correlation is found among other blocks. Global exit predictors can perform well by exploiting this 85% correlation. However, more than half of this 85% of branches is found as predicate-defines in other hyperblocks. The predicate-defines in other blocks contribute to 47% of the total dynamic correlated branches. These are not typically captured by global predictor. However, an exit ID approximately represents the path through the block leading to the exit, and hence, a partial representation of these predicate-defines is provided by the ID. However, this is captured approximately and merge points of different paths and the exit numbering mechanism can impact the level of information captured by the exit ID. Finally, 38% of the correlated branches can be easily tracked by the global exit history registers as they are present as exits in other hyperblocks.

Considering predicate-defines within the same block and within different blocks, more than half of all the strongly correlated branches are present as predicate-defines in the hyperblock code. These results indicate that there is severe loss in correlation among exits in hyperblocks. Sometimes, a branch may be strongly correlated with several other branches and even if some of them are predicated, it could still be predicted accurately if the directions of the non-predicated correlated branches are available. In our work, we have not analyzed the influence of correlated branches by classifying branches that are necessary for the current branch to be predicted correctly and branches that may be if-converted to a predicate-define without loss in accuracy. However, more than 50% of the correlated branches being converted to predicate-defines provides a strong motivation for constructing better predictors using the "lost" correlation information.

5.3 Techniques to use "lost" branch correlation

There have been previously proposed solutions to improve branch predictability when using predicated code. We discuss some of these techniques below and also propose novel software approaches to handle this problem.

Hardware: Including predicate information

If the correlated branch is in the same block or in a different block as a predicate-define, regular branch/exit-history capturing predictors can capture very little of this information through exit IDs. Simon et al. [64, 65] proposed incorporating predicate information in global branch history to reduce the impact of misprediction migration and improve branch predictability. They also determined that using the resolved predicate values from the back-end of the pipeline is better than using the predicted predicate values from a hardware predicate predictor. The accuracy of the predicate predictor determines the effectiveness of using predicted predicate values in the branch/exit predictor.

Hardware: Including a local predictor

A local exit prediction component can track cross-correlation of branches belonging to the same block. This is especially useful when a block does not repeat at short intervals. Hence different dynamic instances of the same static block cannot be captured in the global exit predictor unless the histories are sufficiently long to capture the block instances occurring far apart in time [6, 60]. Consequently, local predictors have to be sized appropriately. This also makes a strong case for the use of local/global tournament exit predictors. A local exit predictor can also include predicate bits from the committed predicate-define instructions in its per-block history when updating the history at commit time. These bits can be used along with the exits while making exit predictions.

Software: Using predictability as a heuristic in the compiler

Improving block construction and predication using profiling techniques have been attempted by several other researchers in the past. Some approaches use block and path frequencies [18, 39] to choose basic blocks or paths for inclusion in hyperblocks while others use branch predictability [40], predictability across input sets [34] and machine learning [72] to pick heuristics for predicating branches in the compiler.

Software: Using branch correlation as a heuristic in the compiler

Previously proposed techniques to improve predication and block formation use information from individual basic blocks like basic block frequency, number of instructions, presence of hazard instructions, edge frequencies, and branch predictability. None of these directly incorporate correlation information across branches to reduce loss of correlation among branches in predicated blocks. Path frequencies collected from path profiling incorporate the behavior of program paths. These frequencies can provide some level of correlation information since branches are frequently highly correlated with other branches in the same path.

Our proposal is to consider correlation among groups of branches to improve block formation and predication. Based on the loss of correlation information we gathered from our analysis, we can profile the behavior of each branch to identify its strong correlation with few other static branches. This can be done for every static branch. The profiled data can be used by the compiler during predication and hyperblock construction.

- If two branches are highly correlated with each other and they are relatively close to each other in static code, the compiler should try to place them as exits in different hyperblocks, so that the correlation information is still available. The next preference would be to place them as exits in the same hyperblock. However, if exit IDs are assigned in such a way as to approximate the predicate path through the block instead of the current program sequential order, they may prove to be more useful in capturing

the predicate correlation and the strategy for intelligent placement of branches as exits instead of predicates may not be necessary.

- If a branch A is highly correlated with branch B, and B is predicated, then it is a good strategy to predicate A also. If a hardware predicate predictor that can feed predicate values to the exit predictor is available, this strategy may not be necessary as the predicted values of predicate-define B can be sent to the exit predictor for prediction of the block containing the exit branch A.

- If correlation information is lost because of predication, the compiler could give hints to the predictor to use the resolved or predicted values from a subset of the predicate define instructions from any block.

More complex scenarios than presented above are possible. For example, misprediction migration might make us consider the correlation of the current branch with a branch which has now been predicated and added as predicate input to the current branch.

5.4 Summary

In this chapter, we analyzed block predictability and correlation and highlighted the inherent difficulty of dynamic global history-based predictors in making predictions for hyperblocks instead of basic blocks. We discussed exit correlation and tried to analyze the reasons for the relative poor performance of exit predictors when compared to branch predictors. A comparison of the prototype exit predictor (5 KB) and a similar 5 KB local/global tournament branch predictor showed that the branch predictor had a 40% lower misprediction rate.

Using interference-free perceptrons to track the correlation between branches in the basic block code and a mapping of the branches in the basic block code to exits and predicate-defines in the hyperblock code, we identified the placement of strongly correlated branches in the hyperblock code. We found that a significant amount of correlation is lost

due to correlation-agnostic hyperblock formation. More than 50% of the strongly correlated branches were present as predicate-defines in other hyperblocks.

We illustrated the importance of the local exit predictor by identifying that 6% of the strongly correlated branches are present as exits within the same hyperblock. These exits can be captured by a local history when it is beyond the window of blocks captured by the global history. We also compared the role of local exit predictor with a local branch predictor and identified that the local exit predictor is more useful to make accurate predictions in a local/global tournament design.

We identified previous work which suggested hardware (making use of predicate values in branch predictor histories) and software techniques (using predictability information in the compiler) to improve predictability. For future work, we proposed correlation-aware hyperblock formation in the compiler that considers correlation among pairs of branches while applying predication and basic block selection for inclusion in hyperblocks. Even though block formation and predication algorithms in the compiler consider several heuristics, correlation is not typically used. We believe that correlation-aware block formation can definitely improve history-based exit prediction and make exit predictors as good as state-of-the-art branch predictors. Another technique to exploit the "lost" correlation in the hardware predictor is to make use of predicate values in the histories or to use the compiler to give exit IDs that capture more meaningful information like the predicated path taken within the hyperblock leading to the exit. These techniques will help include the predicate values/unique path also in the history and have the potential to make exit predictors perform better.

Chapter 6

Distributed Prediction in the TFlex Composable Processor

In the last several chapters, we presented block prediction techniques for TRIPS, a block-atomic distributed architecture with large cores. In this chapter, we explore control flow prediction mechanisms for Composable Lightweight Processors (CLPs) made up of several lightweight cores. We propose distributed block predictors for TFlex, a homogeneous CLP architecture based on an Explicit Dataflow Graph Execution (EDGE) instruction set. TFlex uses the same ISA as TRIPS. TFlex is a fully distributed architecture with no shared structures. TFlex also uses a block-based and block-atomic execution model like TRIPS. This chapter describes the challenges in distributing the block predictor across lightweight cores. We propose a classification of predictors for distributed architectures and evaluate different types of predictors from each category. Finally, we show the effect of distributed predictor accuracy on TFlex performance.

6.1 TFlex core microarchitecture and execution model

In this section we discuss the details of the microarchitecture and execution model relevant to the distributed block predictor. Other details specific to the TFlex architecture can be found elsewhere [30, 31].

TFlex, a fully distributed architecture, has a set of homogeneous lightweight cores in the execution substrate. Each core is a fully functional lightweight processor and has an 8 KB instruction cache, 8 KB data cache, 128 registers, dual-issue capability, an instruction window that can hold up to 128 instructions (one block), and integer and floating-point execution units. Cores can be composed together, if need be, to form a bigger logical processor for high-performance execution. For example, a 32-core TFlex processor consisting of 32 single-issue cores can be employed as an aggressive 32-wide uniprocessor, or 32 lightweight processors to execute 32 threads in parallel, or any combination in between. The TFlex processor uses the TRIPS ISA as well as the block-atomic model of execution [31]. TRIPS and TFlex processors are compared in Figure 6.1. Each core in TFlex consists of the execution core in TRIPS along with a part of the other tiles such as Global Control, Data, Instruction, and Register tiles. Cores communicate through the mesh network connecting the cores. Every core includes an operand-network router. Cores transfer data through the operand network and control information through the global control networks. The cores are sized appropriately to maintain a one-cycle hop latency between cores. Since wire delays can affect performance adversely in distributed architectures like TRIPS or TFlex, the network hop latency should remain as low as possible. Initial experiments with a hop-to-hop operand network latency of more than one cycle showed significant degradation in performance [31]. In the initial TFlex design, the size of each core was estimated as 9.5 mm^2 in a 130 nm process technology [30]. For future technologies, to maintain the latency between adjacent cores as one or two cycles, careful sizing of the cores will be required.

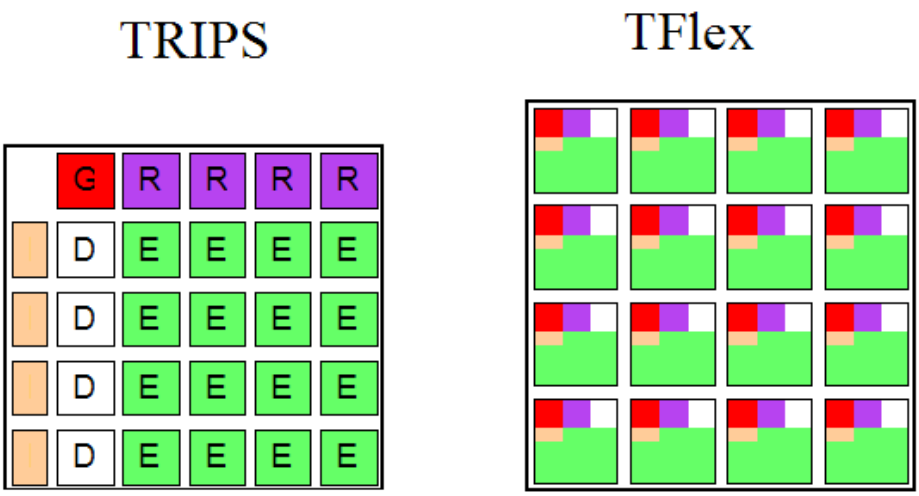


Figure 6.1: Composable Lightweight Processor - TFlex

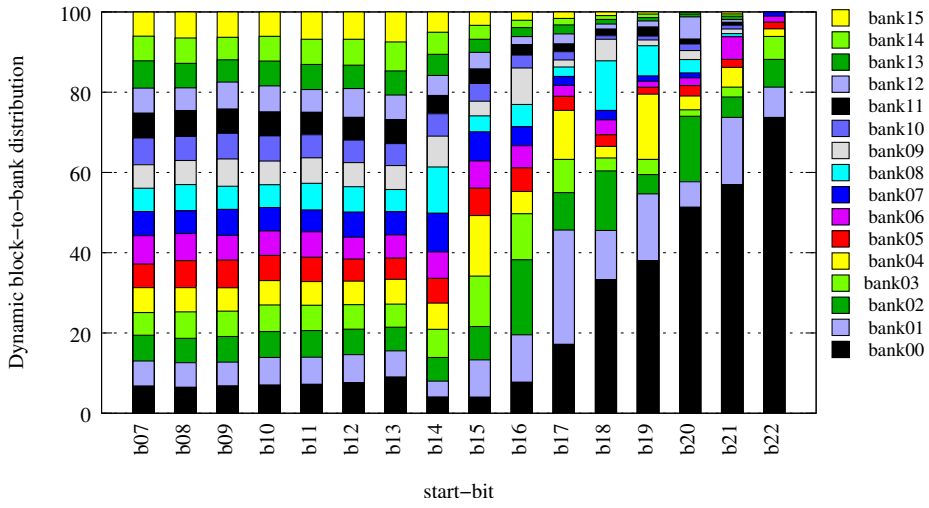
6.1.1 TFlex control protocol

TFlex has a flexible substrate that can operate in several modes depending on the system requirements. In the high-throughput mode, each core can execute a thread. The execution model for a thread that runs on only one core is straightforward. The same core handles all the operations for the core namely fetch, dispatch, execute, and commit. In high-performance mode, a thread can run across multiple cores. There are several possibilities for the control protocol in this mode. One alternative is to specify one core as the control core for all blocks. This core performs all the block control operations including next block prediction. This control model is simple and does not involve distributed control and branch predictor design. The disadvantage of this model is that the control logic and branch predictor state in other cores are not used. Using reduced branch predictor state may lead to reduced prediction accuracies. Using the control unit of one core for all the blocks may lead to control bottlenecks when several blocks are being fetched, executed, and committed across the different cores and the control unit becomes the bottleneck in supporting multiple simultaneous block fetches/executions/commits.

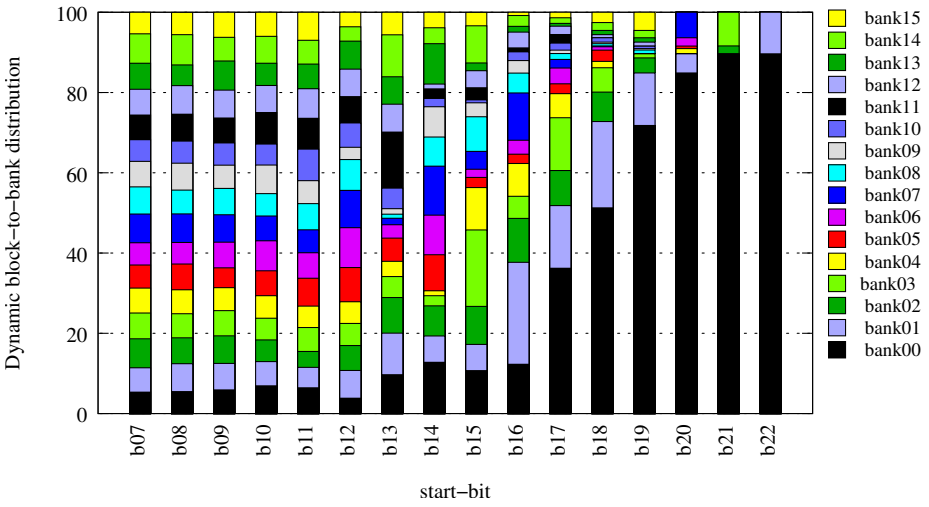
Another alternative for control protocol design is to use a distributed control proto-

col in which different banks perform control operations for different blocks. In this case the protocol may not be as simple as a single-control-core protocol. However, this model has the advantages of being able to use multiple branch predictors across all the cores as well as reducing control bottlenecks by distributing the control for blocks across different cores. In the TFlex architecture, a distributed control protocol is implemented. Each in-flight block is owned by exactly one core. This core is designated as the "owner" for that block. Block ownership is determined by some lower-order block address bits. Hence the ownership is a static distribution which makes every dynamic instance of a static block always have the same owner. This mapping helps achieve cache and predictor locality. The block owner performs all the control operations for the block, namely, sending fetch commands for the block, delivering the next block prediction, sending flush commands on a misprediction of the block, detecting block completion, and finally, committing the block.

Figure 6.2 shows the static block-to-bank distribution for a 16-core TFlex processor and Figure 6.3 shows the dynamic block-to-bank distribution for a 16-core TFlex processor. Four bits from the block address are used to index into the bank when a 16-core configuration is used. The X-axis represents the starting bit position used. For example, *b12* represents that four bits are chosen for the index starting from the 12th bit i.e., bits 12, 13, 14, and 15 are picked as bank index to determine the owner core. The Y-axis represents the fraction of blocks mapping to each core (or bank). The static distribution of mappings for integer and floating point benchmarks are uniform until the 13th and 12th bit positions respectively. Beyond this the distributions are skewed or heavily biased towards cores with lower IDs. The dynamic breakdown shows what percentage of blocks are owned by each core at run time. This number has a direct impact on performance as exploiting locality and uniform access of cores are both important for efficient run time performance. If there is heavy access to a single core, both performance and power/temperature constraints may be affected. Beyond the 9th bit as the starting bit, the floating point dynamic mapping distribution suffers. This may be due to the fact that floating point programs are loop oriented



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

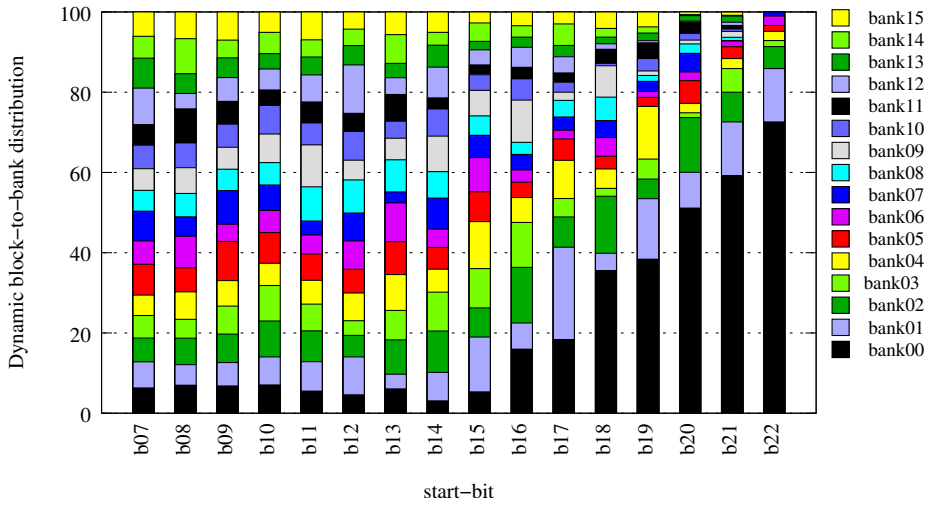
Figure 6.2: Mean static block-to-bank mapping percentages for various bank ownership schemes for SPEC integer and floating point benchmarks for a 16-core TFlex processor. X-axis represents the indexing bits used starting from the bit position marked. For example, b12 represents the four bits picked starting from the 12th bit (bits 12, 13, 14, 15). Y-axis represents the fraction of static blocks that map to each core/bank ranging from bank 0 to bank 15.

with few large blocks executing in the core portion of program. Hence, they repeatedly hit the same bank when the addresses are close to each other. The best distribution is when lower-order bits are chosen. However, for the integer benchmarks, the distribution is almost uniform until the 15th bit. This is because integer benchmarks are not loop-oriented and the complex control flow ensures that several blocks and functions are encountered frequently. Hence, the block addresses are spread apart and they map to several different banks. In all our evaluations we choose the lowest-order bits (starting from *b07*) to give the best uniform mapping to cores for both types of benchmarks.

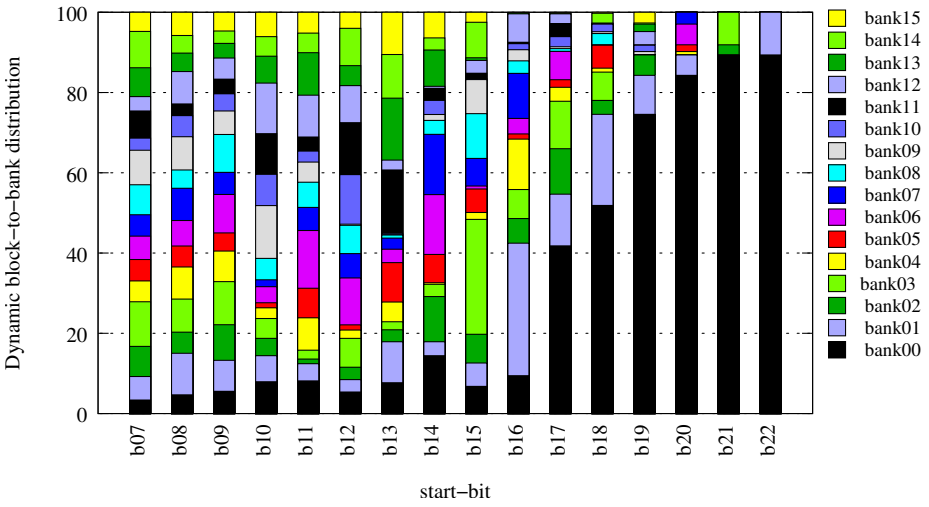
6.1.2 TFlex execution model

In the single-core processor mode, each core can execute one TRIPS block at a time. Each core can fully hold a single TRIPS block in its instruction window. Once this block is fully executed and committed, the next block of instructions can be fetched. When multiple cores are combined to form a single logical processor, several ways of mapping blocks to cores are possible [53]. We use the methodology proposed in [30] in which a block is spanned across all the participating cores. For example, if 16 cores are being used to form a logical processor for a thread, each 128-instruction block will be mapped across all the 16 cores. In this case, each core gets eight instructions from every block that is mapped.

Since TFlex is a highly flexible processor capable of operating in several modes (low or high performance, low or high throughput), control flow predictors for TFlex should be able to predict accurately for any mode efficiently. Specifically, low area, timing, and power metrics are desired. When a thread runs on only one core, the predictor local to the core handles predictions for that thread. If a thread is running on more than one core, then the predictors in all the cores can be used to form a single logical predictor to make predictions for this thread. The TFlex execution model described above designates the responsibility of delivering predictions to the owner core for a block. The owner core has several responsibilities for control flow speculation: making a request for the next prediction, sending the



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.3: Mean dynamic block-to-bank mapping percentages for various bank ownership schemes for SPEC integer and floating point benchmarks for a 16-core TFlex processor. X-axis represents the indexing bits used starting from the bit position marked. For example, b12 represents the four bits picked starting from the 12th bit (bits 12, 13, 14, 15). Y-axis represents the fraction of dynamic blocks that map to each core/bank ranging from bank 0 to bank 15.

predicted next block address to the next owner core, checking for a misprediction when the resolved address is sent from the core that executed the firing exit branch instruction, triggering a flush by sending a flush message to all cores and initiating a predictor recovery if there is a misprediction, and finally, initiating a predictor update when the block completes. When the global control unit in the block owner core sends the block address to the predictor in the same core to get the predicted next address, the predictor unit in the owner core can either make local predictions or request predictions from elsewhere. How the prediction is delivered depends on the specific prediction model used. These models will be described in the following sections.

TFlex is not the first architecture to use a distributed predictor. In the past, researchers have considered distributed predictors for other architectures. Some of the proposals are: the Multiscalar hierarchical predictor [70] and branch predictors for the Core Fusion architecture [19,20]. In Multiscalar, a hierarchical distributed predictor is employed. The top-level predictor predicts the next task to be fetched and mapped on to the next Processing Element (PE) and the individual smaller predictors in the PEs perform branch prediction within the task. The predictors across different PEs use history information from the task predictor also. The PE branch predictors can work in parallel to provide a high overall fetch rate. In the core-fusion architecture individual predictors from different cores are combined together using a monolithic control which maintains common state such as global history and RAS. We compare our approach with these approaches as we describe the distributed predictor in the following sections and in Chapter 7.

6.1.3 Desired features for distributed predictors

Since TFlex is a distributed predictor made up of several small cores (homogeneous CMP with composable cores), the predictor has to meet several requirements on area, power, and latency.

- Predictor area in each core should be small as the cores are lightweight.

- Power dissipation of a single-core predictor should be small.
- Smaller processor configurations built using fewer number of cores have to provide reasonable prediction accuracies without impacting the throughput requirements of the system.
- Predictors for larger processor configurations (high-performance mode) should be highly accurate.
- Communication between predictors from different cores should be minimal or non-interfering with other distributed control communication.

6.2 Distributed next-block prediction classification

A next-block predictor for a CLP architecture like TFlex can be designed in several ways. For now, we assume that *N* cores participate in the execution of a thread and the prediction resources in all the *N* cores can be used to make a prediction, if necessary. An easy way to make a prediction is to choose a single core among the *N* cores and always make a prediction using the block predictor from that core. In this case, the predictor has to be very small, since the cores are lightweight. A small predictor may not give higher accuracies. The predictor resources from other cores are also not used. However, this technique offers a simple predictor control protocol. Another method is to use the predictor in the owner core, every time we make a prediction. Yet another technique is to use all or a subset of the *N* predictors (one from each core) to make a prediction.

We formalize the various ways of prediction described above by proposing the following classification scheme for distributed block prediction. These schemes for a 16-core TFlex configuration are shown in Figures 6.4, 6.5, 6.6, and 6.7.

1. Piecemeal Prediction: The block predictor is distributed across all the cores and only the predictor structures from any one core are used for a block prediction. Piecemeal

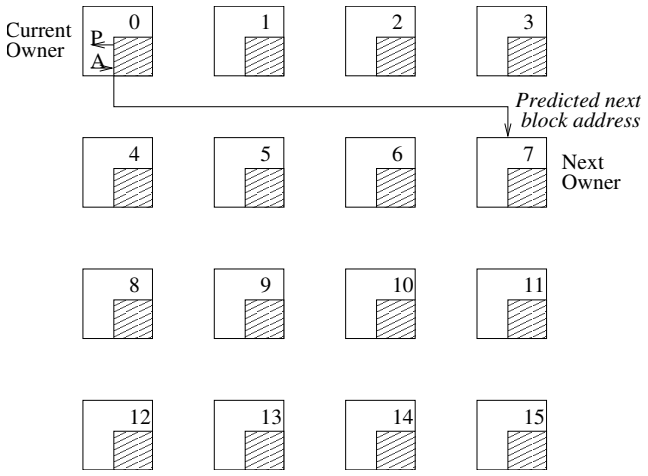


Figure 6.4: Independent distributed prediction shown for a thread running on a 16-core TFlex distributed architecture. *A* represents the block address sent from the control unit of the block owner core to the block predictor. *P* represents the prediction delivered by the block predictor to the control unit of the block owner core.

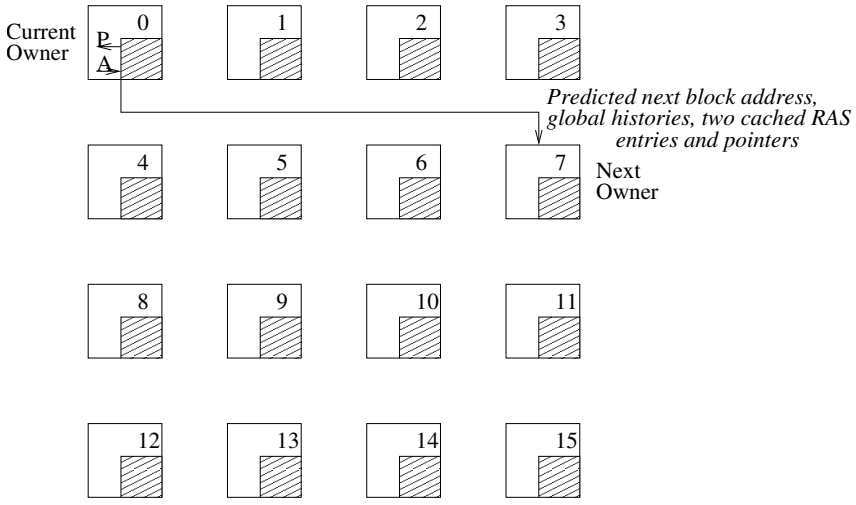


Figure 6.5: Banked distributed prediction shown for a thread running on a 16-core TFlex distributed architecture. *A* represents the block address sent from the control unit of the block owner core to the block predictor. *P* represents the prediction delivered by the block predictor to the control unit of the block owner core.

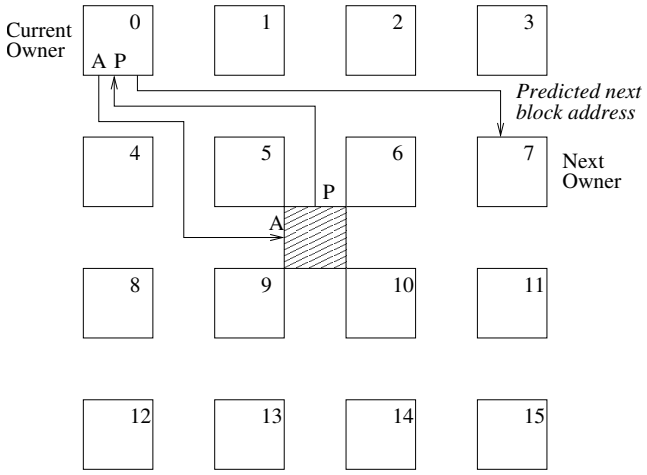


Figure 6.6: Monolithic prediction shown for a thread running on a 16-core TFlex distributed architecture. *A* represents the block address sent from the control unit of the block owner core to the block predictor. *P* represents the prediction delivered by the block predictor to the control unit of the block owner core.

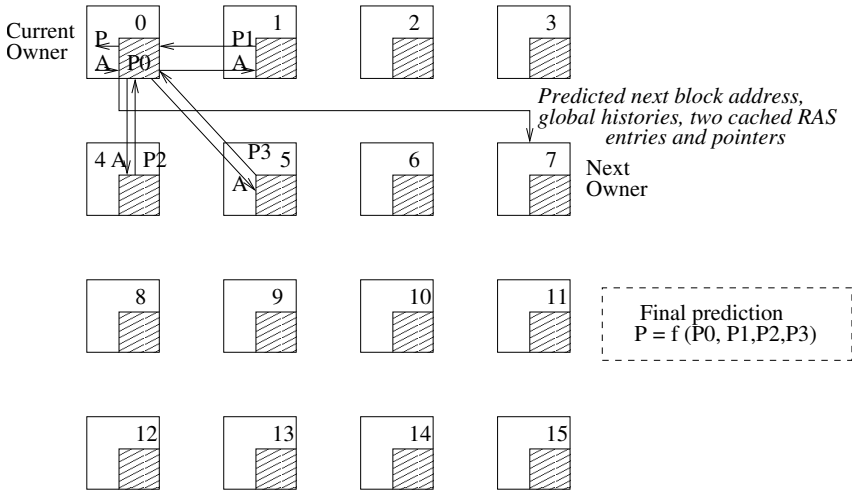


Figure 6.7: Co-operative distributed prediction shown for a thread running on a 16-core TFlex distributed architecture. *A* represents the block address sent from the control unit of the block owner core to the block predictor. *P* represents the prediction delivered by the block predictor to the control unit of the block owner core.

prediction may be of the following two types:

(a) Independent distributed prediction - Every core has a small predictor that makes predictions for the blocks that it owns. The predictor does not communicate with other predictors in other cores.

(b) Banked distributed prediction - Every core has a small predictor that makes predictions for the blocks that it owns. The predictor communicates with other predictor banks in other cores to maintain correlation and other prediction information.

2. Unified Prediction: All or a subset of the active predictors are used to make every block prediction wherever the predictor components are located. Unified prediction may be of the following two types:

   (a) Monolithic prediction - A reasonably-sized predictor situated in one of the cores or outside of the cores which is used to make all the predictions for blocks from a thread. This is a degenerate case of unified prediction.

   (b) Co-operative distributed prediction - A thread running on N cores uses all N cores or a subset of N cores to make a prediction.

In the next few sections we describe each of the design points and evaluate the designs using MPKI, latency, and execution time metrics. For each of these design points we can have several types of monolithic block predictors mapped to a distributed environment. For exit predictors, we evaluate predictors like local, global, local/global tournament, and TAGE-like tagged predictor. We use a multi-component target predictor with Btype, BTB, CTB, and RAS (for some models). In the first TFlex proposal [31], we used a predictor of size 1 KB in each core. The total state in the exit and target predictors put together was approximately 8 KBits. We arrived at this size based on estimates of predictor area and making sure the predictor does not occupy a significant portion of the core area (the 1 KB predictor is approximately 1.65% of a one-core area [30]). We expect that future processors

will require more accurate predictions and slightly scaling the predictor will not have any significant negative impacts on the core area and core-to-core latency. Hence, we choose a 2.5 KB size for the predictor state in each core for all our evaluations. We allocate approximately 1.25 KB to the exit predictor and approximately 1.25 KB to the target predictor.

6.3 Independent distributed prediction

This prediction mechanism is depicted in Figure 6.4. In this class of predictors, every core contains a small full-fledged predictor which is used to make predictions for the block owned by that core. Predictors across different cores are completely independent and do not communicate directly. The global control unit (GCU) within each core requests a prediction from the core's predictor and sends the predicted next block address to the next owner core's global control unit (GCU). Subsequently, the next owner core's GCU requests the next prediction from that core's predictor. Similarly when a block commits, the GCU in the owner core initiates a predictor update in the same core's predictor. When a block misprediction is detected, the owner core initiates a predictor repair operation in the local predictor and broadcasts a flush message to all remote cores participating in the execution of the current thread.

6.3.1 Predictor design

We now describe how the exit and target prediction components can be designed for the independent distributed prediction mechanism.

Exit predictor

The exit predictor in each core has to be small in keeping with the total predictor size requirements (2.5 KB). Scaling down the local/global tournament exit predictor to about 1.25 KB is one option. A local predictor with 128 entries in the first level history table, 256

entries in the predictor table, a global predictor 1K entries, and a choice predictor with 1K entries can be constructed for approximately 1.2 KB. The local exit history table is indexed by the block address. The owner core is also decided using the block address bits. Hence accessing the local core's predictor is expected to give good accuracies as in a monolithic predictor. Whenever the block address is used to calculate the index into predictor tables, the address bits that are used to determine the owner core are excluded, since those bits are the same for all the blocks mapping to that core. The exit prediction and chooser tables (2nd level) are indexed using a hash of the block address and the local/global exit history. To use longer exit histories we can fold the history and hash it with the relevant block address bits. Local histories can be speculatively updated to reflect the latest instances of every block since a given block is always predicted by the same core. However global histories will be stale and incomplete since each predictor only updates the history for the blocks it owns.

Since the global predictor may not be as effective as a monolithic global predictor without up-to-date histories, another alternative exit prediction scheme is to use a local predictor alone. This local predictor can use all the 1.25 KB of storage and hence can be twice as large as the local component in the tournament predictor described above. We can also implement other predictors like GEHL-like or TAGE-like exit predictors which are primarily based on global history by scaling them down to 1.25 KB. However all these multi-component predictors primarily use global histories and the ability to predict accurately is hampered due to incomplete history information. To demonstrate this, we simulate a stand-alone global predictor and compare it with the local and tournament predictors described above.

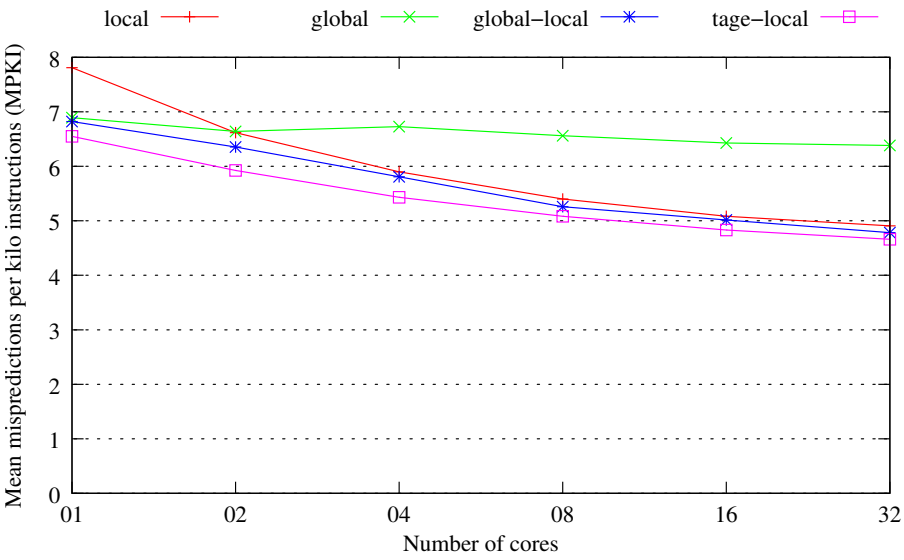
For speculative updates, the speculative structures such as local future file, global history file, and choice history file are included in every core's predictor. For quick predictor repair in the owner core, following a misprediction, these recovery structures are required. Note that each of these recovery structures is indexed using the dynamic block ID. Hence there are as many entries as there can be dynamic blocks mapped to the TFlex processor (for

example, 32 blocks for a 32-core TFlex processor).

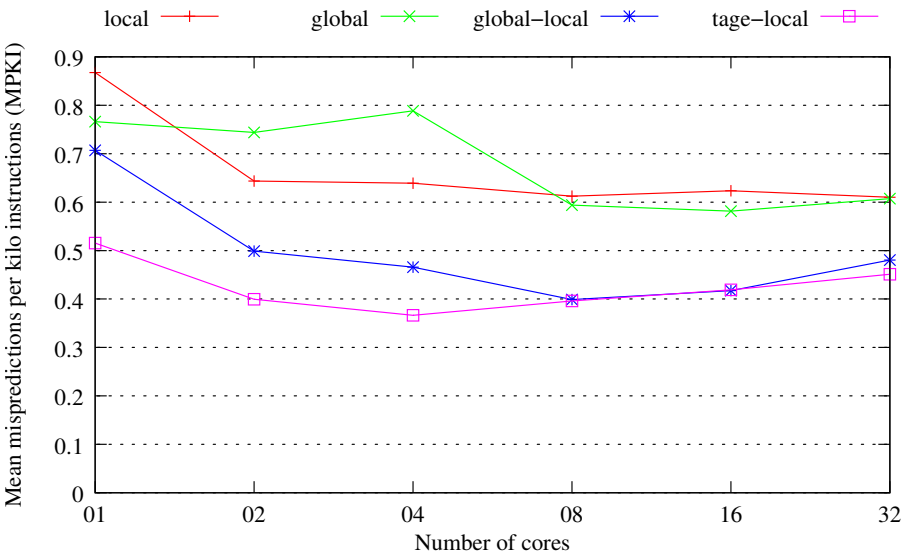
Target predictor

The prototype target predictor consists of the Btype (branch type) predictor, sequential predictor, BTB, CTB, and RAS. To obtain a 1.25 KB target predictor in each core, we reduce the size of all these structures. The Btype predictor, the BTB, and the CTB are indexed using a hash of the block address and the predicted exit. As explained earlier, the block address mapping to cores is static; the owner for a block is determined using the block address bits. Static block address mapping ensures that the predicted exits always get their branch and call target addresses from the same core. Hence implementing these structures is straightforward. The sequential predictor (Seq) consists of logic (adder + control) to find the next block address given the current block address and the block type. The Seq component is included in every target predictor.

To predict returns efficiently, we need a RAS. Having a small RAS in every core and using only that RAS will likely give extremely poor return predictions. A RAS tries to mimic the program call stack and corresponding calls and returns may not map to the same core. When a call is fetched, the predictor may push the return address into core *X* (based on the call block address). When a return is fetched, the predictor may try to pop the return address from the RAS in another core *Y* (based on the return block address). Our compiler does not try to ensure the mapping of corresponding calls and returns to the same core. Hence we exclude the RAS from the target predictor. Instead, we use the BTB to predict return addresses also. A BTB may not be very accurate for most returns. However, for returns in functions being called only from a single call point, a BTB should be able to learn the return addresses almost all the time. One negative impact of this model is the higher aliasing in the small BTB.

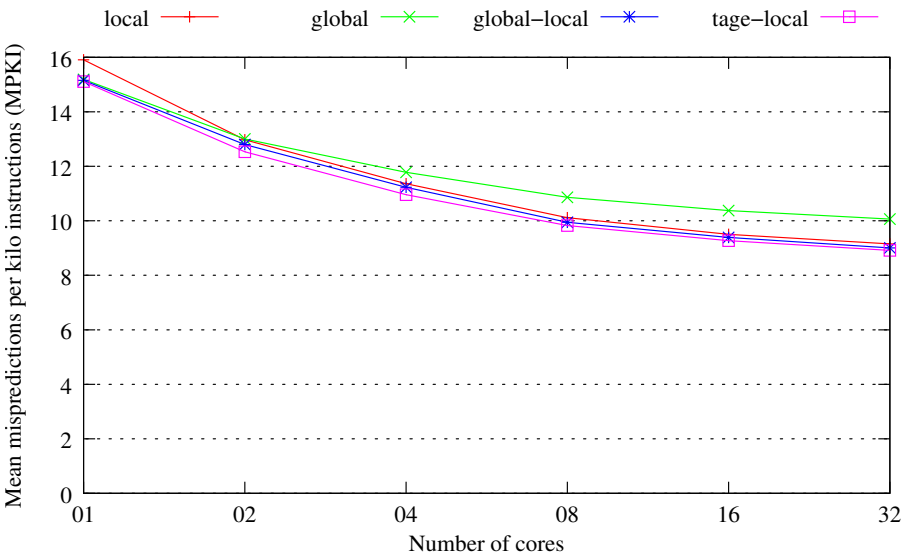


(a) SPEC2K integer benchmarks

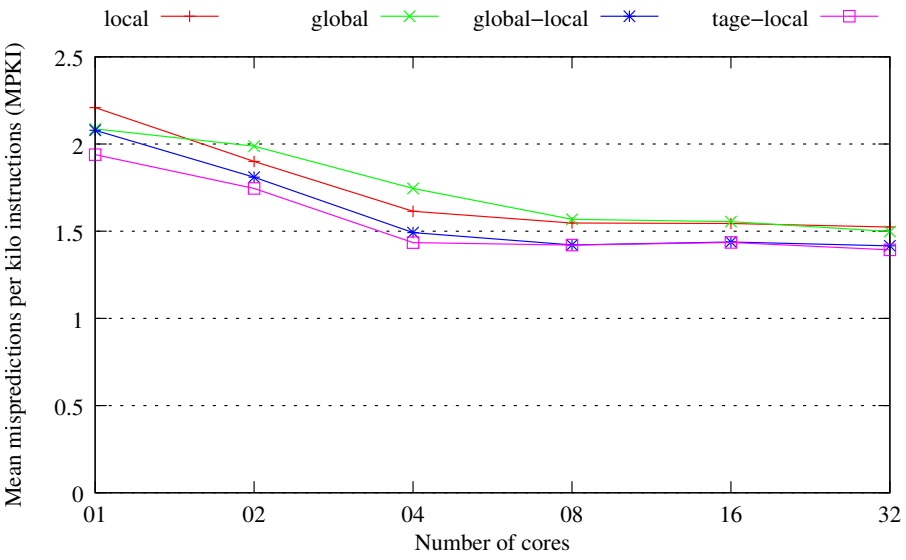


(b) SPEC2K FP benchmarks

Figure 6.8: Independent distributed prediction - Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean exit MPKI is shown for four predictors (local, global, local/global tournament, and local/TAGE tournament) predictors for integer and FP benchmarks.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.9: Independent distributed prediction - Overall MPKI for 1 to 32 core TFlex core configurations. The mean MPKI is shown for four predictors (local, global, local/global tournament, and local/TAGE tournament) predictors for integer and FP benchmarks.

6.3.2 Predictor evaluation

We evaluate independent distributed predictors with four types of exit predictors: local predictor, global predictor, local/global tournament predictor, and local/TAGE tournament predictor. The target predictor remains the same for all evaluations. Figure 6.8 shows the mean exit predictor MPKI for the four predictors we simulated. We evaluate six TFlex configurations from 1 to 32 cores. When a thread is running on only core, the local predictor (7.81 MPKI) performs the worst, followed by the global predictor (6.89 MPKI). The tournament predictors are the best for a one-core configuration with the local/TAGE predictor performing better (6.55 MPKI) than the local/global predictor (6.82 MPKI). When using more cores, the global predictor does not scale well while the other three predictors improve steadily. This is because the global predictor makes predictions based on global histories and they need to be shared to capture all the correlation. The local predictor is banked the same way as the block-to-core banking and so the MPKI reduces steadily as the number of cores increases. However, the local predictor is always slightly worse than the tournament predictors (which are able to predict some more branches correctly due to the presence of the global component). For a 32-core configuration, the local, global, local/global, and local/TAGE predictors have MPKIs of 4.91, 6.38, 4.78, and 4.66 respectively. Figure 6.9 shows the overall MPKI for the four types of predictors. We found that overall MPKI is significantly higher than the exit predictor MPKI due to the large number of return mispredictions. The difference between the global predictor and the other predictors is less pronounced. Other than the global predictor, the rest of the predictors have almost the same overall MPKI. For FP benchmarks both the graphs show that the two tournament predictors are better than the local an the global predictors.

6.4 Banked distributed prediction

This prediction mechanism is depicted in Figure 6.5. In this class of predictors, every core contains a small full-fledged predictor which is used to make predictions for the block owned by that core. Predictors across different cores can communicate to help improve the overall prediction accuracy of the running thread. The global control unit (GCU) within each core requests a prediction from the core's predictor and sends the predicted next block address to the next owner core's global control unit (GCU). It can also send other prediction information to the next owner or to other cores. This information can be used by other cores when they make subsequent predictions. Similarly, other cores can send messages to the current owner core which it can use to make current or future predictions. We call this prediction approach "banked" because the predictors in individual cores can be considered to be banks of a single large predictor. At any point in time, one core's banks are accessed for exit and target predictions. The banks share information with each other for making better predictions.

We use a simple model for communication which aims to reduce the number of messages communicated between different predictor banks. An owner core requests predictions from its local predictor unit and sends the predicted next block address to the next owner core. Along with the next block address and fetch commands send to the next owner, we send the speculatively updated global and path histories, if available, to maintain accurate histories. This technique helps ensure that the next owner sees the speculatively updated histories. Multiscalar used hierarchical predictors in which the higher level task (block) predictor forwards its task histories to the next processing element (PE). The PE uses this history as a starting predictor history to do branch predictions within the task. The idea is that this provides some relevant context and correlation information for the branches with a task. When branch predictions are being made within a PE, the branch prediction outcomes are shifted into the predictor history. In our model, we do not have multiple histories working at the same time at different hierarchies. We use a single level global/path history which

is updated and sent from core to core. At any point in time the owner core has the latest history while the other cores that are not currently making predictions have stale histories.

During a branch misprediction, the owner core repairs its histories and sends the repaired histories to the next owner core (determined using the resolved branch address). During a completion/commit time update, the owner core updates all its tables.

6.4.1 Predictor design

We now describe exit and target prediction designs for the banked distributed prediction mechanism.

Exit predictor

We first describe a scaled-down version of the prototype exit predictor and then describe some other possible predictor designs. The prototype exit predictor consists of a local/global tournament predictor and choice predictor. The local predictor design is similar to the design described for the independent distributed predictor. The local prediction table uses longer histories and folds them to generate indices with the appropriate length. We use history folding to make use of longer histories even while using smaller tables. Folding also has a disadvantage. Folded histories can lead to more destructive aliasing in the prediction tables.

The global and the choice predictors both have global exit history registers in their first level. These histories are speculatively updated with the predicted exit and the updated histories are sent to the next owner core along with the predicted next block address. Even though at any point in time only the owner core makes predictions, we need to have these exit histories in every core so that the latest histories are available as soon as they are needed during a prediction request. The second level tables for the global and the choice predictors are indexed using the address and history hashed together just like the local exit prediction table hashing. When viewed globally, the second level tables use a combination

of the Gselect and Gshare schemes (proposed by McFarling in [41]). This is because banking based on block address bits gives a natural Gselect like index (concatenation of block address bits with history bits to form second level table index). The XOR-ing of the remaining lower order address bits with the history bits within each core's predictor provide the Gshare-style index.

Speculative updates structures are similar to the structures in the independent distributed predictor design described in the previous section. All structures holding speculative checkpoint and recovery state are present in every core's predictor. Throughout all our designs, we use a simple approach of allocating full-sized speculative structures. For example, in a 32-core TFlex substrate, a maximum of 32 cores can be allocated to a thread. These 32 cores together can execute at most 32 blocks in parallel. Hence we use 32 entries (indexed by the dynamic block ID) for the local future file, and global/choice history files.

Other types of banked distributed exit predictors can be designed similar to the distributed local/global tournament predictor described above. To implement a GEHL-like predictor, a small scaled down version consisting of a small number of prediction tables and history lengths are used. The global histories are sent from one core to another as in the tournament predictor. Similarly, a TAGE-like predictor can also be implemented. Since the exit predictor in each lightweight core has to be small, multi-component predictors such as OGEHL and TAGE can have only few components to make predictions. For example, it will be difficult to construct a 7-component TAGE predictor using only 1.25 KB in storage.

Target predictor

The target predictor is a scaled-down version of the TRIPS prototype multi-component target predictor. We use all the components present in the prototype target predictor. The Btype predictor, Sequential predictor, BTB, and CTB are designed similar to the components in the independent distributed predictor. The challenging part of the distribution is in

designing the RAS and the links to establish call-return relationships as in the case of the monolithic TRIPS prototype predictor. We first list the key steps in the call-return prediction in the monolithic predictor (presented in detail in Chapter 2 and shown in figure 6.10) to help understand distributed call-return prediction:

1. Call Fetch: During a (predicted) call fetch, retrieve the call target and return address from the CTB. Send the call target to the fetch engine and push the return address on to the RAS.

2. Return Fetch: During a (predicted) return fetch, pop the return address from the RAS and send is to the fetch unit.

3. Call commit: Update call target in CTB and push CTB index (link) and call block address into the RAS link stack.

4. Return commit: Pop the CTB index (link) and call block address from the link stack, index into the CTB with the index and store the return address offset in the CTB entry.

Small RAS structures need to be present in every core but when multiple RAS components are put together, they should work together as a single stack. We propose a distributed RAS that is sequentially partitioned across all cores. All the participating cores for a thread together form the logical RAS for that thread with physically distributed stack entries. A core that is currently making the next prediction needs information about the current location of the top of RAS stack (core ID) and the top of stack value. Calls and returns send messages to update the stack top in the appropriate core. In addition, they also send the updated top of stack value to the next block owner core (just as the global histories are sent) so that the next owner will have the top of stack information when it makes a prediction. This communication avoids additional penalty in fetching the top of the stack from a different core in case the next block fetched has a return branch. We require the following structures for return prediction in each core:

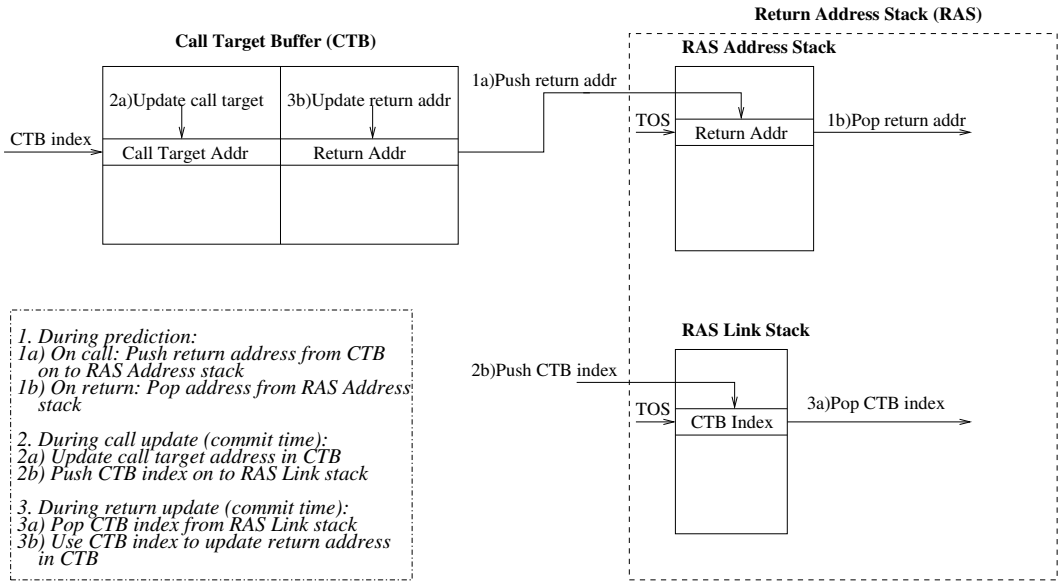


Figure 6.10: Call-Return mechanism in the TRIPS prototype predictor

- Return Address Stack (RAS): Each core contains a small stack and the stacks from all the cores form the larger logical stack. The bottommost of the stack is assumed to be in the core with ID *0* in the logical processor for a thread. Hence, if the logical processor uses *N* cores, the stacks from cores 0, 1, ... *N-1* in sequence form the logical larger RAS.

- Link Stack (LS): LS is split similar to RAS. In addition each entry also includes the call core ID to indicate where the call corresponding to the return was mapped.

- Address Stack Pointers and Link Stack Pointers (ASP and LSP): We need these stack pointers for each core for the predictors to be able to determine the stack top.

- RAS History File (HF): A history file in every core with as many entries as the maximum dynamic blocks allowed (32 for a 32-core TFlex processor) is required for quick misprediction recovery.

- Each core needs to know how many cores are participating in the execution of the current thread to reset the stack pointer to zero after it reaches the maximum value.
- Each core needs four extra registers to hold the current two top of stack values for both the address as well as the link stacks (called TOS0 and TOS1).

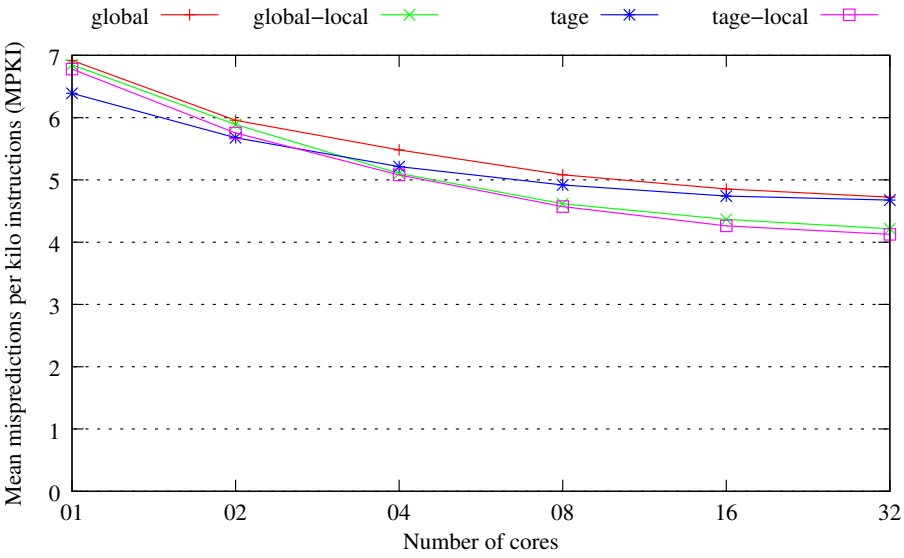
Using these new structures, we implement the following distributed call-return prediction mechanism described in the following steps.

1. Call fetch: Read call target and return address from CTB; send call target (which is the predicted next block address) to the predictor control in the local GT; locate RAS top core and send return target to that core; TOS1 = TOS0, TOS0 = return address; send updated TOS values to next owner.
2. Return fetch: Send TOS0 as predicted return target; TOS0 = TOS1; send request to stack top to update stack and also request to send the new second top to the next owner core.
3. Call commit: Update call target in CTB. Push CTB index, call block address and call bank on to the LS by sending a message to the stack; LS TOS1 = TOS0, TOS0 = return (index, address, bank); send updated stack values to the next owner core.
4. Return commit: Retrieve TOS0 and send message to bank containing CTB to update the entry with the return address offset; decrement pointer and send message to LS stack top to update itself; send the updates to the next owner core.
5. All transactions are tagged by a time stamp to order events at the stack top.
6. Misprediction Recovery: The owner core retrieves the stack state before the mispredicted block was fetched using its local RAS history file contents and sends messages to the core containing the stack top location as well as the next owner core.

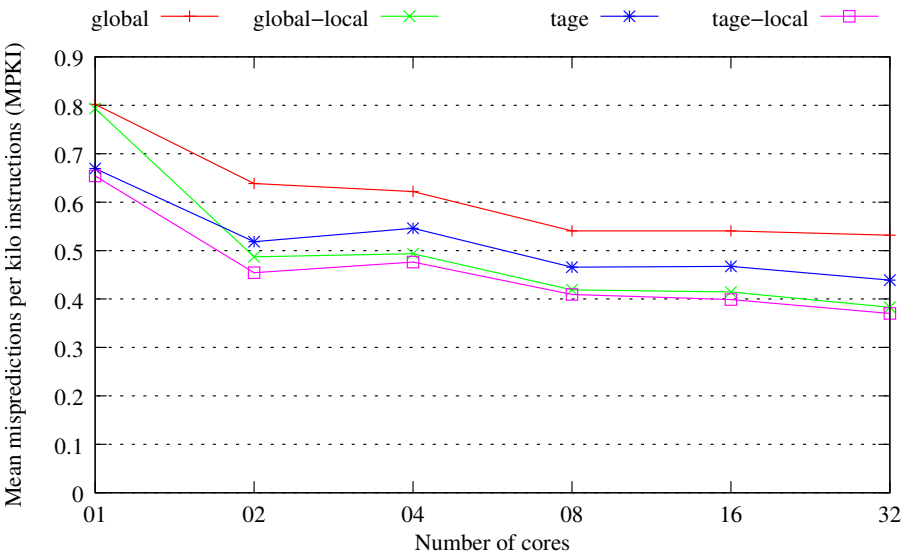
In this distributed RAS design, there are more messages sent during every RAS prediction or update. However the latency of predicting returns is not high as the top two stack entries are cached in the special TOS registers in the current owner core. Since update timing is not as crucial as prediction timing, this design will not have timing drawbacks. It is possible that messages get reordered or arrive late due to control network traffic. Since returns are infrequent compared to regular direct branches in SPEC integer benchmarks, the penalty of increasing the delay for return prediction may not be high. However for object-oriented codes containing several small functions, the number of calls and returns may be high leading to frequent return address predictions. Hence, ensuring that returns can be predicted without excessive delay due to control network congestion is important.

6.4.2 Predictor evaluation

Predictor results are shown in Figures 6.11 and 6.12 for exit and overall MPKI respectively. Four predictors are shown: global, local/global tournament, TAGE, and local/TAGE tournament. The key difference between this class of distributed predictors and the independent distributed predictors is that the global history, RAS entry, and RAS pointer information can be shared across cores. Local predictor designs for the independent and banked predictors are identical as they do not need any history sharing across cores. Hence, we do not show local predictor MPKI in these graphs. The global predictor has a large improvement in the MPKI when moving from an independent to a banked design. For example, in a four-core configuration the MPKI reduces from 6.72 for independent prediction to 5.48 for the banked prediction. There is reduction in MPKI for the local/global and local/TAGE predictors also. The third bar for each benchmark shows the distributed TAGE predictor with a size of 1.25 KB in each core (using 4 tagged tables) and the fourth bar shows the local/TAGE tournament predictor. The best predictor when using very few cores is the TAGE predictor. It is extremely efficient in using the predictor space and avoiding unnecessary replication while preventing aliasing due to the use of tags. For a one-core configuration

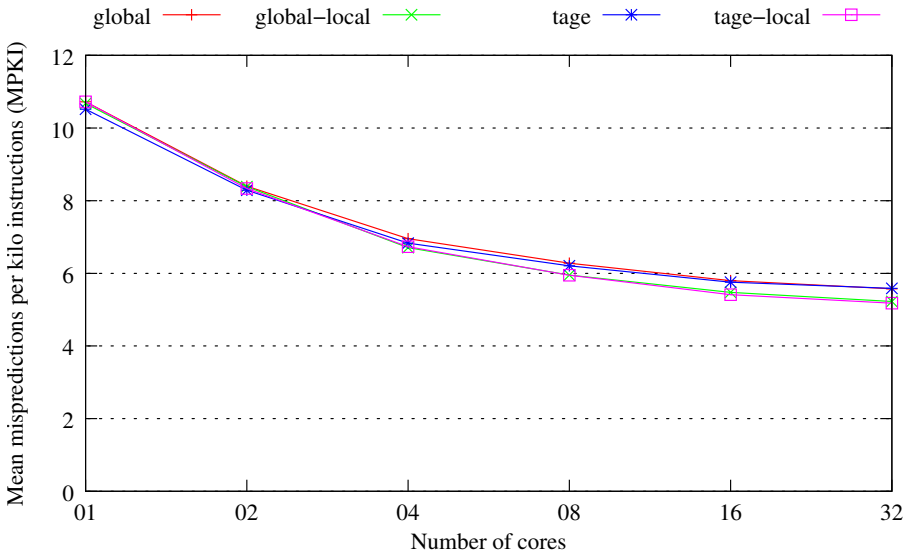


(a) SPEC2K integer benchmarks

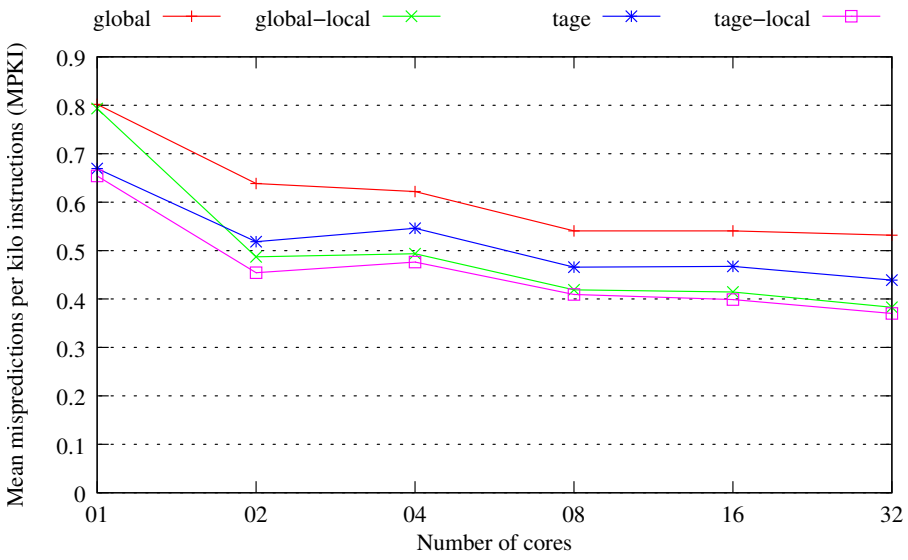


(b) SPEC2K FP benchmarks

Figure 6.11: Banked distributed prediction - Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean exit MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.12: Banked distributed prediction - Overall predictor MPKI for 1 to 32 core TFlex core configurations. The mean MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.

the TAGE predictor achieves an overall MPKI of 6.39 (against a 6.78 MPKI by the second-best local/TAGE) and for a two-core configuration the TAGE predictor achieves an MPKI of 5.68. Beyond two cores, however, the best predictors are the local/global and the local/TAGE tournament predictors. For the FP benchmarks also, the tournament predictors offer the lower exit and overall MPKI values.

The overall MPKI values for the banked predictors are much lower than the overall MPKI for the independent predictors. For a 32-core configuration, the target misprediction rates improve by around 42% for the integer benchmarks and by more than 73% for the FP benchmarks compared to the independent predictors. This is mainly due to the inclusion of a distributed RAS and reducing the number of RAS mispredictions.

6.5 Unified monolithic prediction

This prediction mechanism is depicted in Figure 6.6. Unified predictors make predictions by using all of the predictor state in all cores or a subset of the state in all cores. In either case, predictors from various cores work together to make a prediction for every block. The monolithic predictor is not a true distributed predictor but we include it in our evaluation as it has a simple design and fits our definition of unified prediction. All the predictor state is present in one core. This can either represent all the state for predictors in the TFlex processor or it can be the only enabled predictor unit in the processor. For example, the other predictor units in other cores may be turned off for power savings.

The advantages of the monolithic predictor are the simple design and potentially high accuracies it can deliver (due to all the state being accessible in a single predictor unit with longer histories). The drawbacks are that the latency for retrieving each prediction may be quite high depending on the location of the current owner and the location of the monolithic predictor. Secondly, handling multiple requests may be a little difficult for a processor that is more aggressive than TRIPS. For example, one owner core may request a flush, one core may request an update and another core may request a predict. Multiple

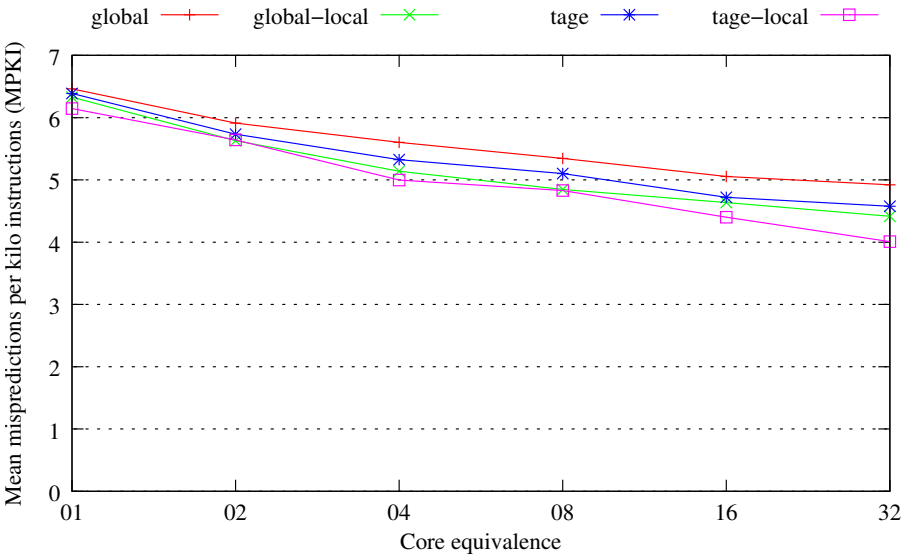
ports are required (with high latency and area penalties) or serializing accesses is required to solve this problem. Serialization can make reduce the prediction rate. For example, in a 16-core configuration, it takes only two cycles to fetch all 128 instructions in a block (each core can fetch four instructions every cycle). In such a scenario, a slow predictor that takes more than two cycles to make a prediction will be the bottleneck in the fetch pipeline. In general, for high-performance execution, throttling fetch because of predictor latency leads to significant performance degradation.

6.5.1 Predictor evaluation

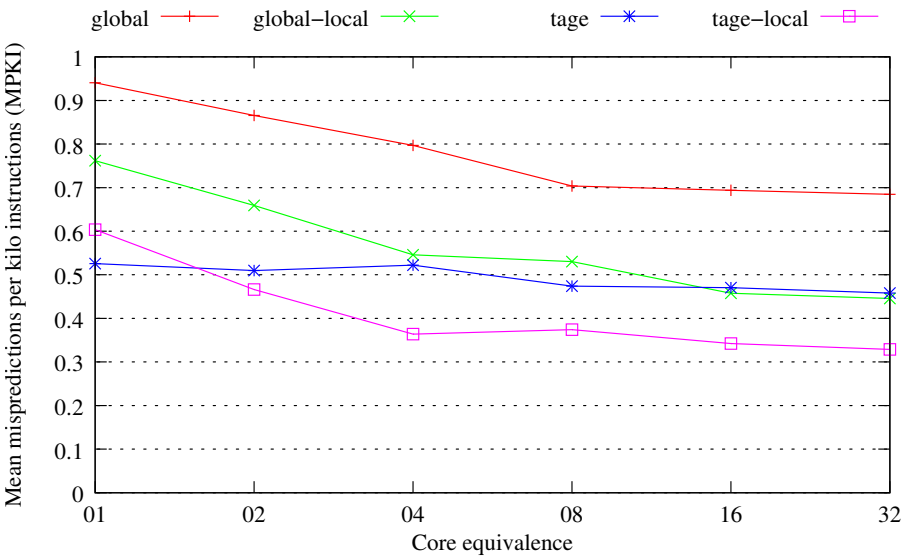
For the monolithic predictor, we evaluate the same four predictors as in the banked design:: global, local/global tournament, TAGE, and local/TAGE tournament. We plot the MPKI for 1 to 32 cores. For each core configuration we give 2.5 KB times the number of cores for the monolithic predictor. This experiment serves to show the comparison between monolithic and other distributed predictors in other models. A 32-core configuration is assumed to have an 80 KB predictor in one core or somewhere between cores in the center of the processor. . This is difficult to implement as it would be large compared to the size of a single lightweight core. Hence, this experiment has been performed for a direct predictor accuracy comparison with other distributed predictors.

Results are shown in Figures 6.13 and 6.14 for exit and overall MPKI respectively. When considering the exit MPKI, the local/TAGE tournament predictor is uniformly the best for all the core counts for integer benchmarks and configurations with more than one core for the FP benchmarks. While the local/global tournament closely follows the local/TAGE for integer benchmarks, the TAGE closely follows the local/TAGE for FP benchmarks. The global predictor is consistently the worst performing exit predictor in these configurations.

The overall MPKI graphs show that the four predictors are very close in predictor MPKI for integer benchmarks. For the FP benchmarks the local/TAGE predictor is the

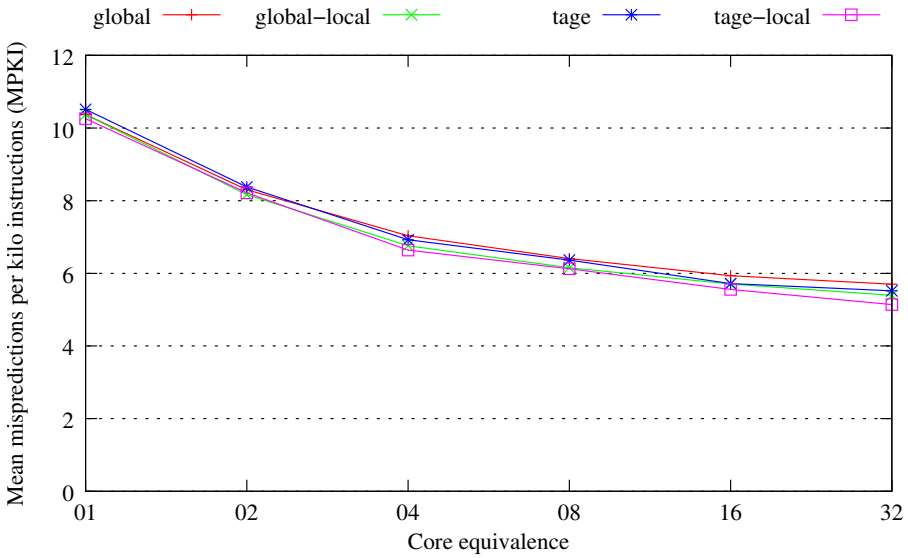


(a) SPEC2K integer benchmarks

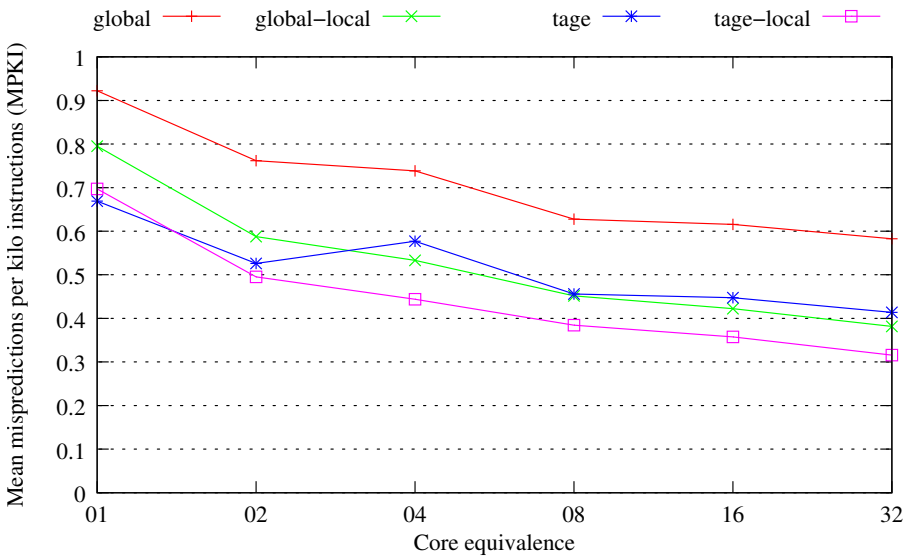


(b) SPEC2K FP benchmarks

Figure 6.13: Unified monolithic prediction - Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean exit MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.14: Unified monolithic prediction - Overall MPKI for 1 to 32 core TFlex core configurations. The mean MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.

clear winner. For both suites, the predictors scale well when the core counts (which implies predictor storage) are increased. The tournament predictors with the monolithic design have slightly higher exit MPKIs compared to their same-sized counterparts from the banked distributed design. This is because, for simpler predictors like the global or the local/global tournament predictor, the use of Gselect-like indexing helps in removing aliasing and this effect compensates for the loss in correlation due to reduced history lengths used to index into smaller tables.

6.6 Co-operative distributed prediction

This prediction mechanism is depicted in Figure 6.7. In this class of predictors, a prediction request from the control unit of the owner core can be satisfied by predictors from several cores. All predictor components from all participating cores may be used for prediction. Another alternative is to use the predictors from a subset of cores to make the prediction. For example, an owner core with core ID *0*, may use its own predictor along with predictors from cores with ID *1* and ID *2* to determine the final prediction. This type of co-operative prediction can be used for exit as well as target prediction. Target prediction is usually simple for branches and calls because the tables are indexed using the address bits and they are naturally partitioned as in the independent and banked prediction schemes described earlier.

In the independent and banked distributed approaches, exit prediction is not very efficient due to the use of small table indices, small history lengths, and using the Gselect approach along with the Gshare approach instead of the Gshare approach alone. However one advantage is that some aliasing may be reduced due to the banking (which provides the Gselect-like behavior). Our inspiration for the co-operative prediction idea is based on recent advances in branch prediction that use several history lengths and several prediction tables to predict the branch direction. In Chapter 4 we proposed exit predictors inspired from state-of-the-art branch predictors. The co-operative distributed idea stems from the

fact from that such predictors which use several histories and tables can be naturally partitioned across multiple cores. For example, allocating one history and one prediction to each core, and computing the final result after collecting the predictions from each of the cores is a solution to easily partition the predictor structures. This solution tries to eliminate the table aliasing and small history factors proposed earlier. The disadvantage is that when several cores are combined together the prediction latency can be quite high as all the cores need to deliver the predictions to the owner core before it can make a final decision.

There are several branch predictors that can be naturally partitioned as described above. We construct distributed predictors using the global predictor, tournament predictor, and the TAGE-like exit predictor. Consider four cores participating at a time. The global predictor represents a degenerate case of unified prediction. When a block is to be predicted by the owner, it sends the block address and latest history to all the other three participating components. Each of the four components participating in the current prediction are assumed to have a portion of the global history table indexed using the Gshare-style index as an XOR of the branch address and the history. All of the components attempt to access such an entry from their table, but only one of them will have the entry corresponding to the index. That component will send the predicted exit to the owner core (or if that component is the owner, nothing needs to be sent). The worst case time to predict will be twice the maximum number of hops (which is two if the four cores are in a square arrangement) for transfer of data and the additional predictor latency for prediction in any of the cores. The tournament predictor can be partitioned similar to the global predictor described above. The global and the chooser predictors are partitioned in a similar way while the local predictor is always accessed from the owner component. The components deliver the global and choice predictions to the owner core. The owner selects between its local prediction and the received global prediction using the choice predictor.

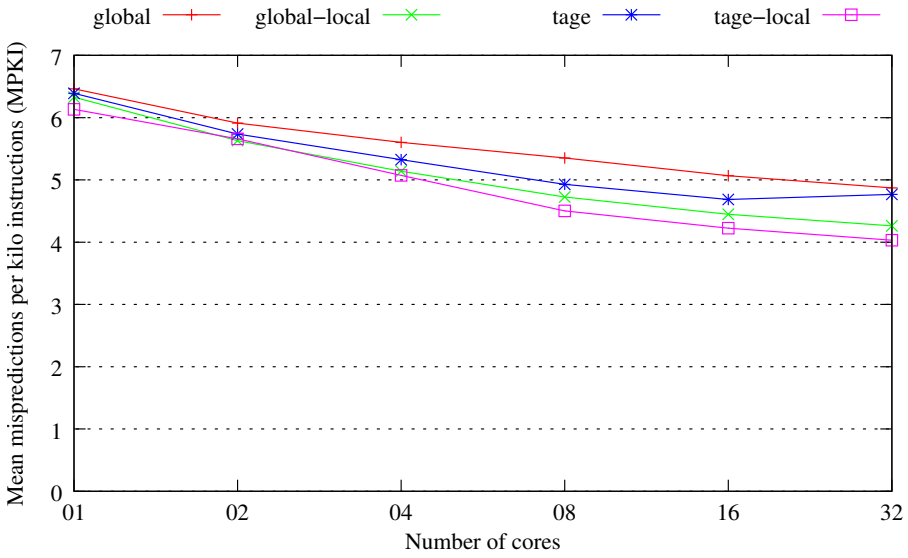
For the TAGE-like predictor we can partition the tables in two ways. One method is to use a multiple of four tables and allocate the same number of tables to each of the four

participating cores. In this case the tables must be equal in size to maintain homogeneous predictor structure. If, for example, we have only a four-component TAGE predictor, all the four tables will have equal size with one table in each core. Since the first component in a TAGE predictor is typically a tagless bimodal predictor (to be used as the default prediction when there is no tag match), the tags allocated to this component will be left unused. However, a TAGE-like exit predictor performs well with unequal partitions when the last table and the bimodal table are given the maximum size and the rest of the tables are given smaller sizes. To use this design with unequal table sizes, we can partition each of the tables in the same way as described above for the global predictor. For example, a 3-component TAGE predictor with one 1 KB bimodal, one 1 KB 10-bit history, and one 2 KB 20-bit history predictor components can be partitioned into four identical parts with each core getting 0.25 KB of the bimodal components, 0.25 KB of the 10-bit history component, and 0.5 KB of the 20-bit history component. This scheme does not waste the tags in the bimodal component. Each of the table indices are generated by all the participating cores. However exactly one of the cores will find a match for index to the bimodal table. Similarly exactly one core will find a match for the index into the 10-bit table. All the three indices may match on the same core or different cores. Once all the predictions are collected (whether a tag was hit, and if hit, the prediction made), the owner core can arrive at a decision for the final prediction by looking at the tags that matched and deciding on the longest-history matching component.

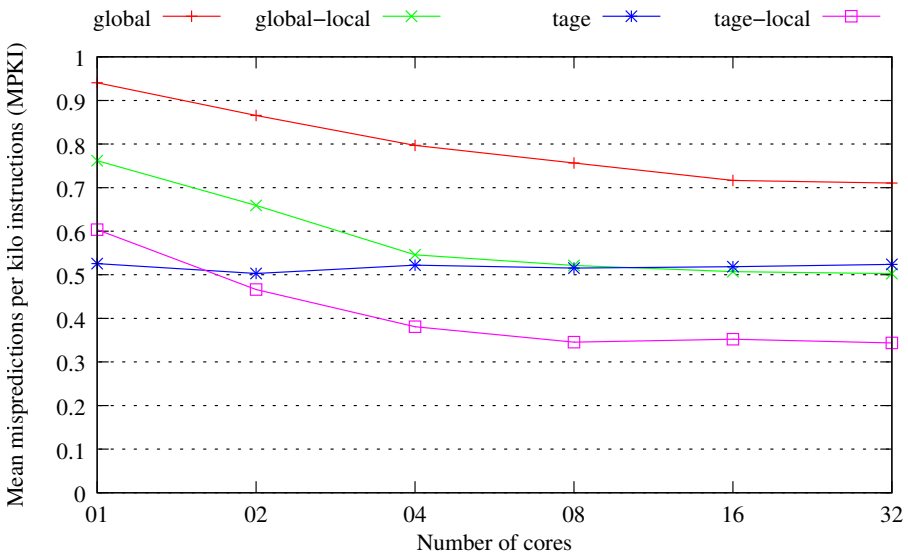
6.6.1 Predictor evaluation

We evaluate global, local/global tournament, TAGE, and local/TAGE cooperative predictors for 1 to 32 cores. The global predictor represents a degenerate case wherein the predictions may come from any participating core but only one core delivers the predictions at a time. Results are shown in Figures 6.15 and 6.16 for exit and overall MPKI respectively.

The exit MPKI numbers show that the local/TAGE exit predictor is the best among

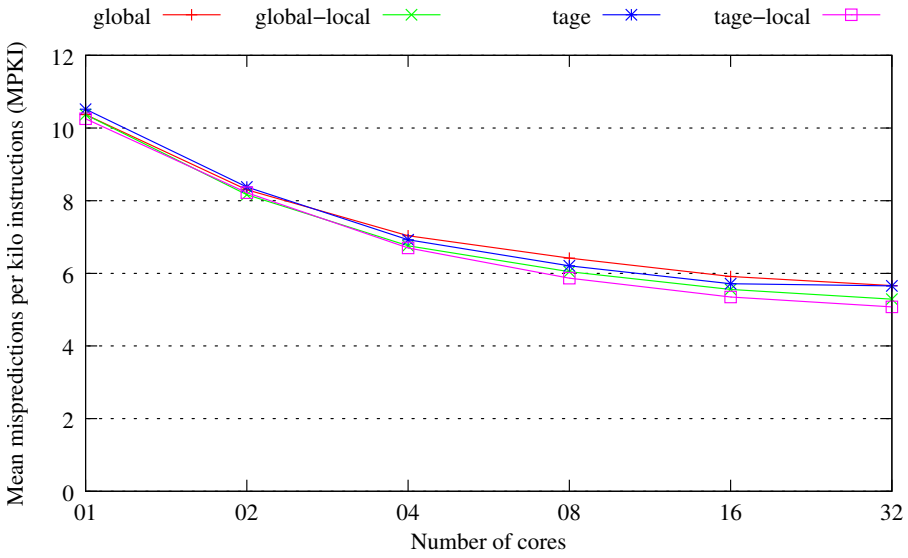


(a) SPEC2K integer benchmarks

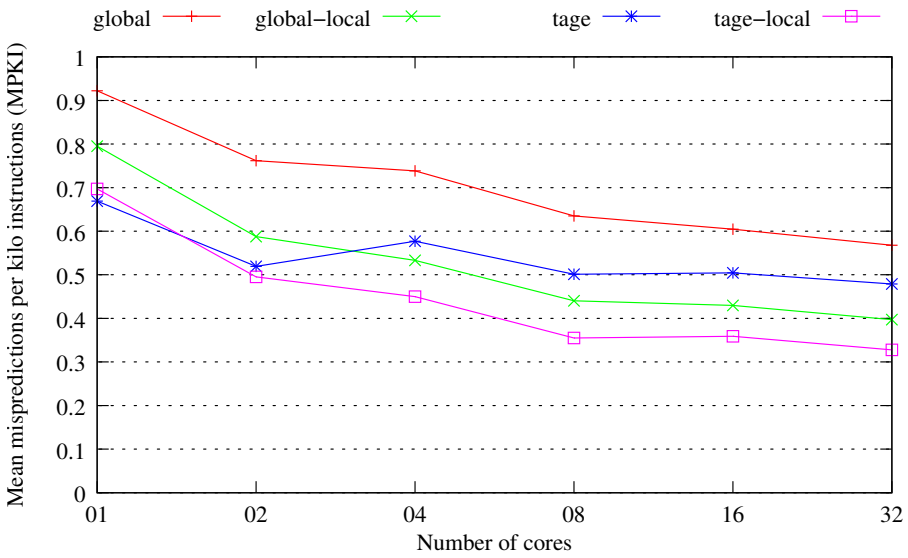


(b) SPEC2K FP benchmarks

Figure 6.15: Unified co-operative distributed prediction - Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean exit MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.16: Unified co-operative distributed prediction - Overall MPKI for 1 to 32 core TFlex core configurations. The mean MPKI is shown for four predictors (global, local/global tournament, TAGE, and local/TAGE tournament) predictors for integer and FP benchmarks.

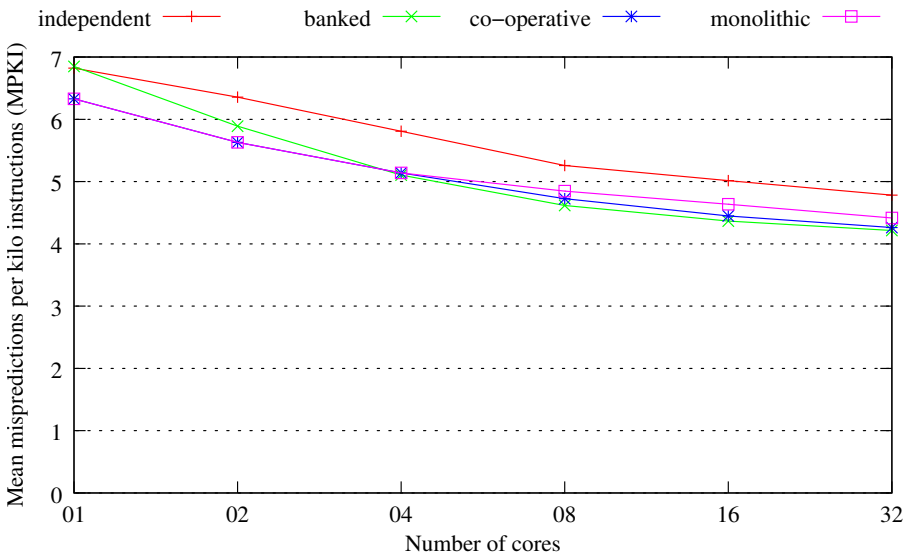
the four predictors. The MPKI values are slightly lower than the monolithic predictor's MPKI. The overall MPKI results also show a similar trend like the banked distributed predictor: the local/TAGE is the best predictor followed by the local/global predictor. The local/TAGE predictor's exit MPKI goes down from 6.13 to 4.03 from 1 core to 32 cores for the integer suite. The overall MPKI for the integer suite goes down from 10.25 to 5.0, showing a more than 50% reduction in the mispredictions when scaling from one-core to 32-core TFlex. For the FP suite, we see a more than 52% improvement from one-core to 32-core configuration.

In general, the accuracies of the banked distributed predictor and the co-operative distributed predictor are comparable and slightly superior to the monolithic predictors. We also found that uniformly, the local/TAGE predictor performs well. However, when considering a large number of cores like 16 or 32, the local/global tournament predictor also performs very well. If the design requirements need a simpler implementation, the local/global tournament predictor can be chosen.

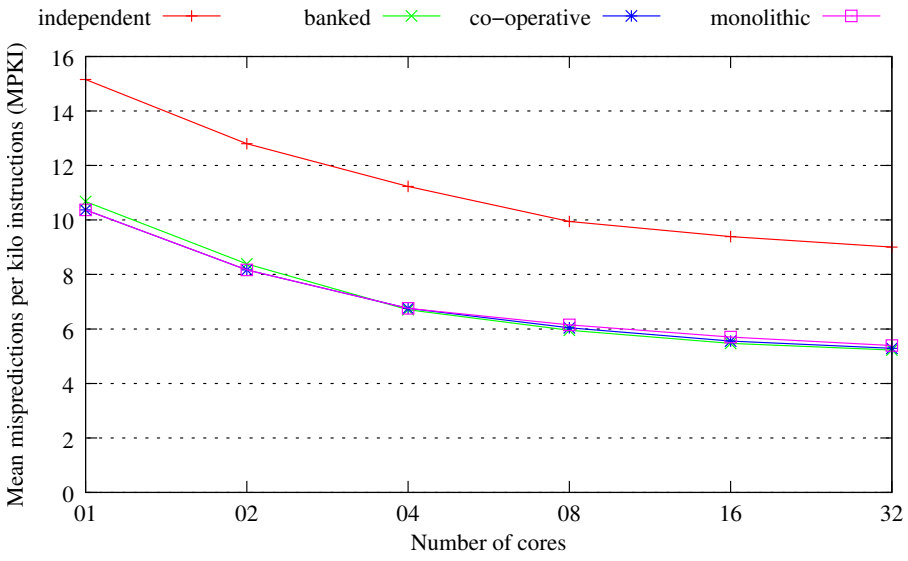
6.7 Comparison of four approaches to distributed prediction

In this section we compare the above described four approaches using two of the best predictors evaluated in the previous sections. We choose the local/global tournament predictor and the local/TAGE tournament predictor for our comparison experiments. We evaluate the integer benchmarks only since the MPKI values for the FP benchmarks are quite low. The floating-point MPKI results were presented in the previous sections. We present exit MPKI and overall MPKI results for six TFlex configurations from one-core TFlex to 32-core TFlex.

Figure 6.17 shows the mean integer exit and target MPKI for independent, banked, monolithic, and co-operative predictor with a local/global tournament predictor. All the four types of prediction strategies scale well when the number of cores is increased. When considering exit MPKI, for small configurations like the one-core TFlex and two-core TFlex,

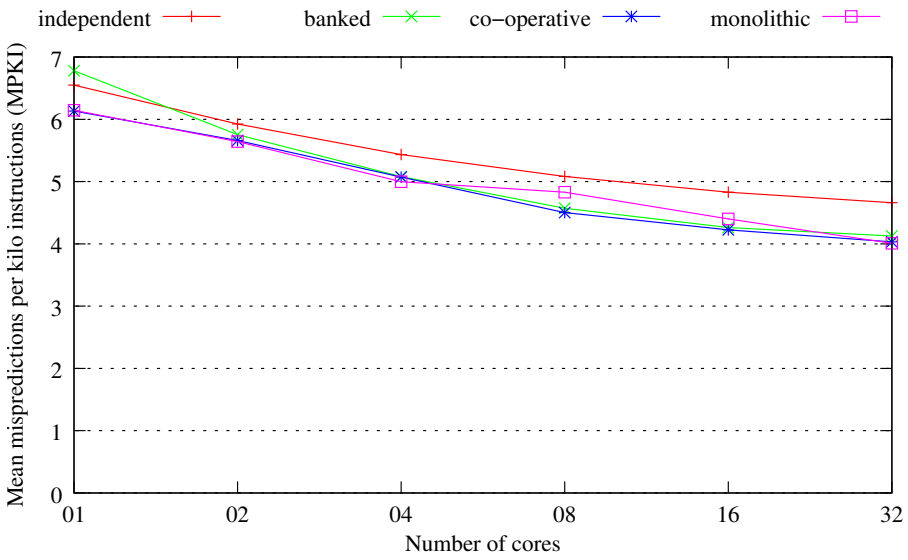


(a) Exit predictor MPKI

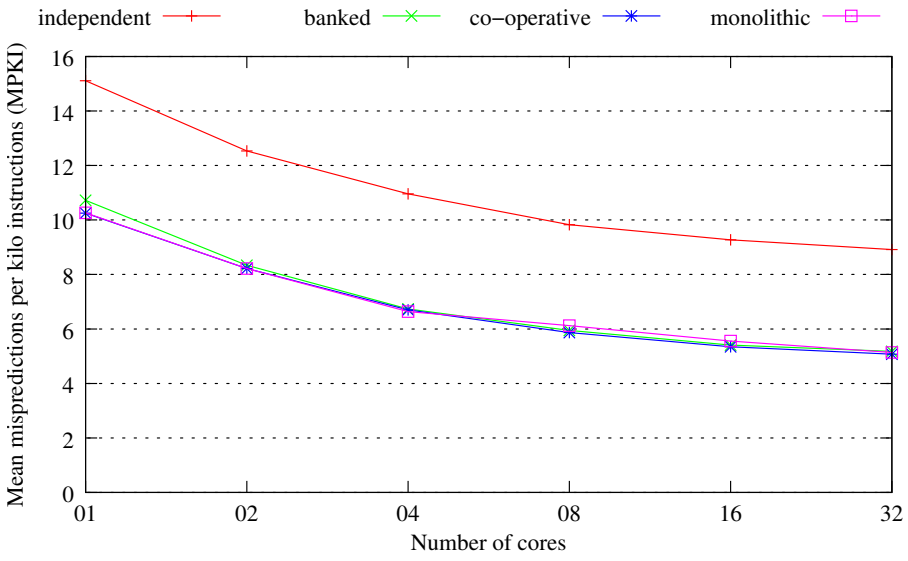


(b) Target predictor MPKI

Figure 6.17: Comparison of mean MPKIs of independent distributed prediction, banked distributed prediction, monolithic prediction, and co-operative distributed prediction for one-core to 32-core TFlex configurations. The top graph shows the exit MPKI and the bottom graph shows the target MPKI. Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean integer MPKI is shown for the local/global tournament predictor.



(a) Exit predictor MPKI



(b) Target predictor MPKI

Figure 6.18: Comparison of mean MPKIs of independent distributed prediction, banked distributed prediction, monolithic prediction, and co-operative distributed prediction for one-core to 32-core TFlex configurations. The top graph shows the exit MPKI and the bottom graph shows the target MPKI. Exit predictor MPKI for 1 to 32 core TFlex core configurations. The mean integer MPKI is shown for the local/TAGE tournament predictor.

the best configurations are the monolithic predictor and the co-operative predictor. For larger configurations the best predictors are the banked distributed predictor and the co-operative distributed predictor. The banked approach works best for four cores or larger configurations. The independent distributed predictor has the lowest exit MPKI among the four predictors due to the poor performance of the global predictor component which does not share histories across cores. When considering the overall MPKI, we find that the banked distributed predictor and the co-operative predictor perform as well as the monolithic predictor for all core counts. However, the performance of the independent distributed predictor is significantly worse when compared to the other cores. The reason for the poor performance is the global component providing lower MPKI as well as the absence of the Return Address Stack.

Figure 6.18 shows the results for a local/TAGE tournament predictor for the four categories of distributed predictor designs. As for the local/global tournament predictor, all the four predictors scale well when the number of cores is increased. The trends are mostly similar to what is observed for the local/global tournament predictor. In the case of the piecewise independent predictor though, the exit MPKI is much lower compared to the exit MPKI of the independent predictor in the local/global tournament predictor. The overall predictor MPKI shows a much larger difference between the independent predictor and the other predictors. This is similar to the previous local/global design as the target predictor remains the same for both designs. The independent predictor not having a RAS results in several target mispredictions. The banked distributed design and the co-operative distributed design achieve nearly the same MPKI as the monolithic design.

In general, the overall MPKIs achieved by the local/TAGE tournament predictor are lower than the MPKIs from the local/global tournament predictor. For a one-core configuration, the best predictor is the local/TAGE predictor in the co-operative design with an MPKI of 10.25. For a two-core configuration the best predictor is the local/global predictor with an MPKI of 8.16 in the monolithic and the co-operative configurations. For the

four-core configuration, the best predictor is the local/TAGE monolithic predictor with an MPKI of 6.64. For the eight-core, 16-core, and 32-core configurations, the best predictor is the local/TAGE predictor with a co-operative design with MPKIs of 5.87, 5.35, and 5.07 respectively. On the whole, the best performing predictor is the local/TAGE tournament predictor with a co-operative design.

6.8 TFlex performance evaluation

In this section we evaluate the performance of the TFlex processor with various styles of predictors. We show Instructions per Cycle (IPC) and execution time speedups for the committed instructions that are useful for the program execution. Mispredicated instructions are not counted. When counting mispredicated instructions, we count instructions which were nullified because of a non-matching predicate as well as instructions that were executed but did not lead to the production of outputs (for example, instructions which target nullified instructions leading to nullified outputs). Moves are also ignored as they are mainly used for dataflow execution support (when a data value has to be targeted to several instructions on the execution substrate).

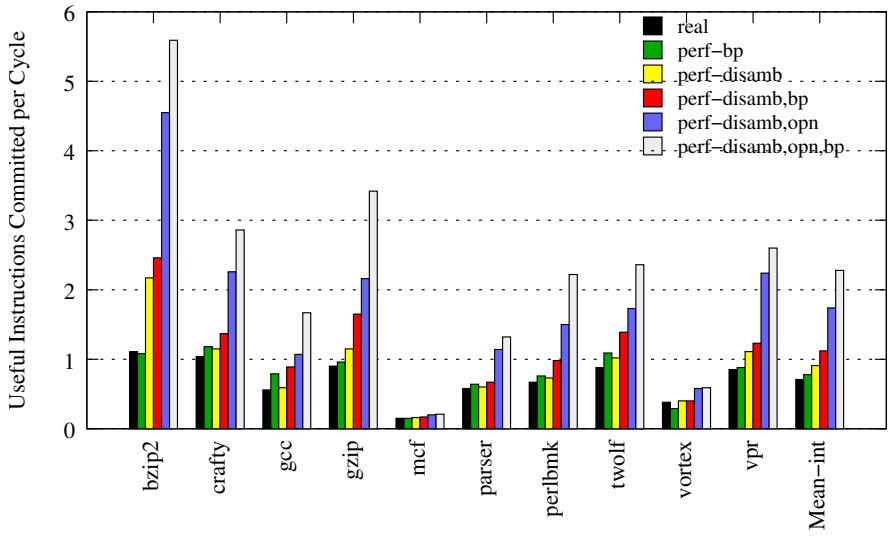
6.8.1 TFlex performance with a banked distributed predictor

Performance results for the integer and floating-point suites are shown in Figure 6.19 for a 16-core TFlex configuration with a banked distributed local/global tournament predictor. Each benchmark has six bars showing the true useful instructions committed per cycle (IPC). The first bar shows the IPC of a realistic 16-core TFlex configuration and the second bar shows the IPC of the 16-core TFlex processor with perfect block prediction. The third and fourth bars show the IPCs of the 16-core TFlex with perfect disambiguation with realistic block prediction and perfect block prediction respectively. The fifth and sixth bars show the 16-core TFlex IPC with perfect disambiguation and perfect OPN with realistic block prediction and perfect block prediction respectively. Hence each pair of bars show

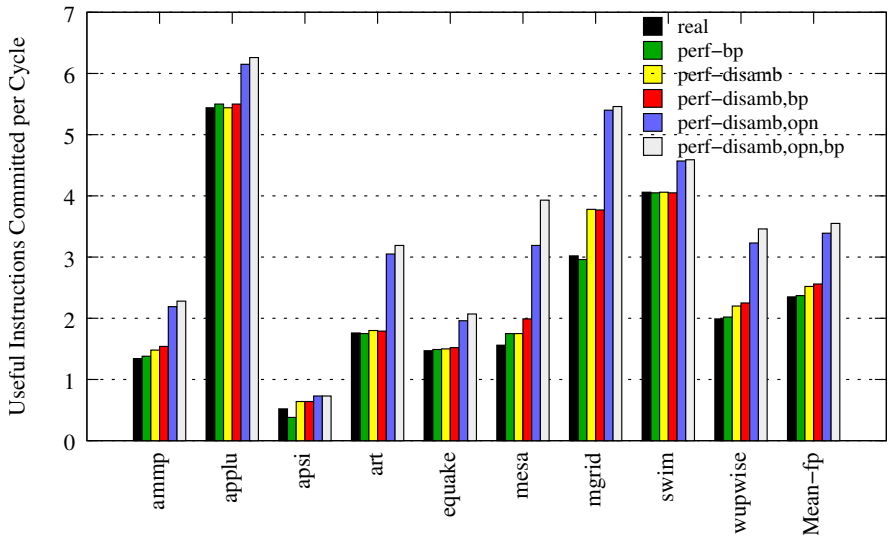
the IPC with realistic and perfect block prediction for three TFlex configurations (realistic, perfect disambiguation, perfect disambiguation and perfect OPN). Figure 6.20 shows the speedups obtained by each of the above configurations over a baseline 16-core TFlex model. These speedups are represented using three stacked bars (each bar showing real and perfect block prediction configurations) with the same groupings as discussed above (realistic TFlex, with perfect disambiguation, and with perfect disambiguation and perfect OPN).

For the integer benchmarks, the mean IPCs from Figure 6.19 shown in the first two bars representing the realistic TFlex configuration are 0.71 and 0.78. This indicates that by making the block prediction alone perfect, we can get a performance improvement of 9.9% on average (from Figure 6.20). This is a much lesser potential improvement compared to performance with perfect block prediction for conventional processors. We also observe that the absolute IPCs are less than 1.0 for all benchmarks except *bzip2* and *crafty* in the realistic configuration. This configuration uses a reasonably good local/global tournament predictor with a banked distributed design. The reason for the lower IPC numbers is the presence of other bottlenecks in the architecture. For some benchmarks like *bzip2* and *vortex* we see a reduction in IPC even when using a perfect block predictor because of microarchitectural effects like higher load-store violations and load-store queue overflows in perfect block prediction mode. The challenges in the baseline TFlex include block prediction, memory disambiguation, operand network contention and latency, and cache and memory latency/bandwidth in the microarchitecture. Hyperblock construction and instruction placement by the compiler also have significant effects on performance. There has been considerable work in improving various components in TFlex [30, 53, 54, 59]. Further improvements can come from microarchitecture design optimizations and better block construction in the compiler.

We consider two of the main bottlenecks to performance for our evaluations: memory disambiguation and operand network contention. The third to sixth bars in the graphs show the effect of perfect branch prediction after removing other main bottlenecks for per-

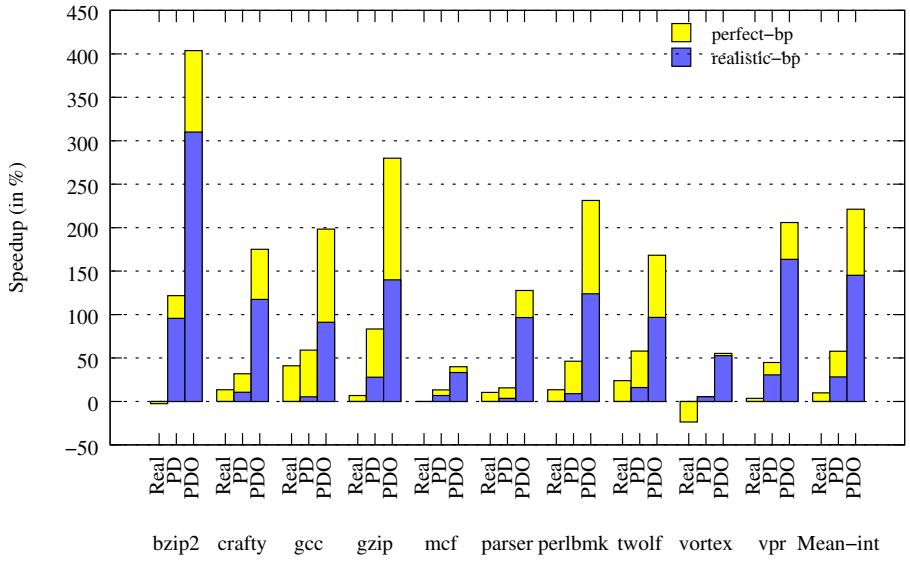


(a) SPEC2K integer benchmarks

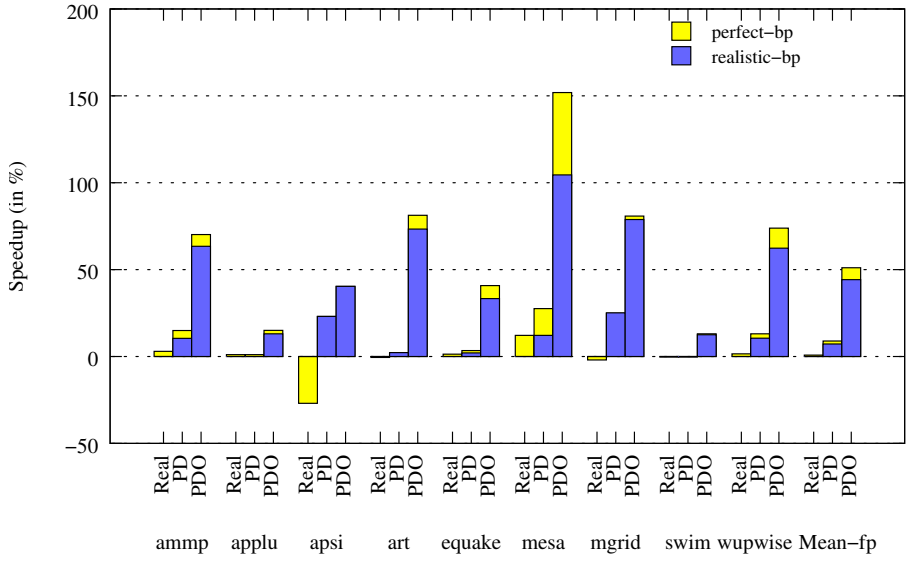


(b) SPEC2K FP benchmarks

Figure 6.19: Performance (IPC) of SPEC integer and floating-point benchmarks with various TFlex configurations. The first bar and second bars compare realistic and perfect block prediction respectively, for a realistic TFlex configuration. The third and fourth bars compare realistic and perfect block prediction respectively, for a perfect-disambiguation TFlex configuration. The fifth and sixth bars compare realistic and perfect block prediction respectively, for a perfect-disambiguation and perfect-OPN configuration. All results are for the 16-core TFlex configuration. The exit predictor is a local/global tournament predictor.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.20: Execution time speedup of SPEC integer and floating-point benchmarks with various TFlex configurations (realistic - *Real*, perfect disambiguation - *PD*, and perfect disambiguation and perfect OPN - *PDO*) comparing the relative speedup of each configuration over the baseline all-realistic TFlex configuration. Each of the three stacked bars show the realistic block prediction and perfect block prediction speedups (stacked) over the realistic TFlex configuration. All results are for the 16-core TFlex configuration. The exit predictor is a local/global tournament predictor.

formance. The third bar and fourth bar compare realistic block prediction and perfect block prediction with the memory disambiguation made perfect. The IPCs using realistic and perfect block prediction with a perfect memory disambiguation configuration are 0.91 and 1.12 respectively. The performance improvement in this configuration with perfect block prediction is 23.1%. Finally, removing the OPN contention bottleneck also, along with the memory disambiguation we get an improvement of 31% for the perfect-prediction, perfect-disambiguation, and perfect-OPN configuration (with IPC of 2.28) over the real-prediction, perfect-disambiguation, and perfect-OPN configuration (with IPC of 1.74). The performance potential of improving the block prediction is significant for TFlex when other more severe bottlenecks are removed. The IPCs are also much higher. Compared to a realistic TFlex, using perfect disambiguation results in 28.2% speedup. Making the memory disambiguation as well as OPN contention ideal pushes the speedup a lot with over 145.1% increase in performance. Except for some memory-bound benchmarks like *mcf* and *vortex*, most other benchmarks show a significant increase in IPC as the bottlenecks are removed. These results also show that once some other microarchitectural components are optimized, the baseline TFlex performance will be much higher and the emphasis on better block prediction will be stronger. This is because the performance loss due to block mispredictions is higher when the baseline processor is performing well.

For FP benchmarks, the mean IPCs for a 16-core TFlex for the six configurations from left to right are 2.35, 2.37, 2.52, 2.56, 3.39, and 3.55 respectively. With everything else realistic, moving to a perfect block predictor enables an improvement of less than 1% in performance. This is mainly because FP programs are not difficult to predict due to their loop-oriented behavior with large loop counts and due to the presence of memory and network bottlenecks. The performance improvement due to perfect prediction when removing the memory disambiguation bottleneck is marginally better (1.6%). When removing the disambiguation and OPN contention bottlenecks by making them ideal, perfect block prediction results in 4.7% increase in performance. The performance improvement for the

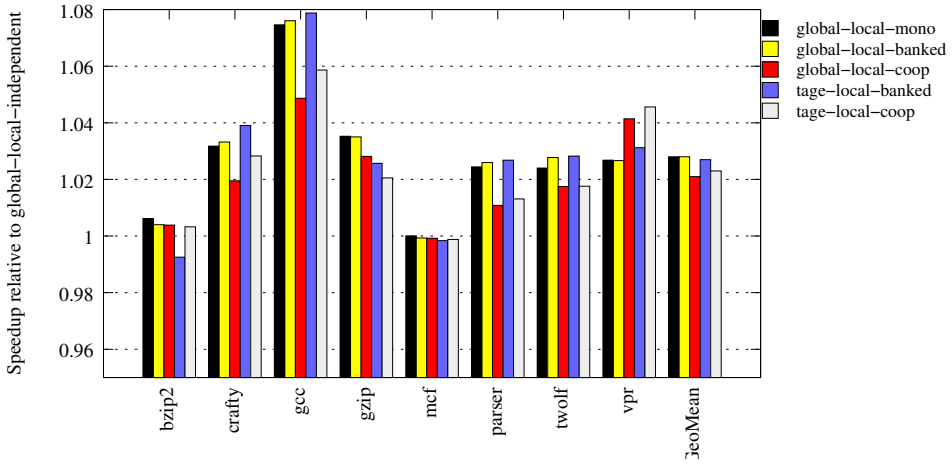
perfect disambiguation mode over a baseline TFlex is only 7.2% while the performance improvement from the perfect disambiguation and perfect OPN over a baseline TFlex is 44.3%. On the whole, there is less potential for higher performance in the floating-point benchmarks when considering perfect block prediction or perfect memory disambiguation. However, since the majority of floating point code is dominated by arithmetic operations, the operand transfer latency is extremely important which is indicate in the performance increase in the fifth and sixth bars where the configurations have no OPN contention.

6.8.2 Performance comparison of four distributed prediction approaches

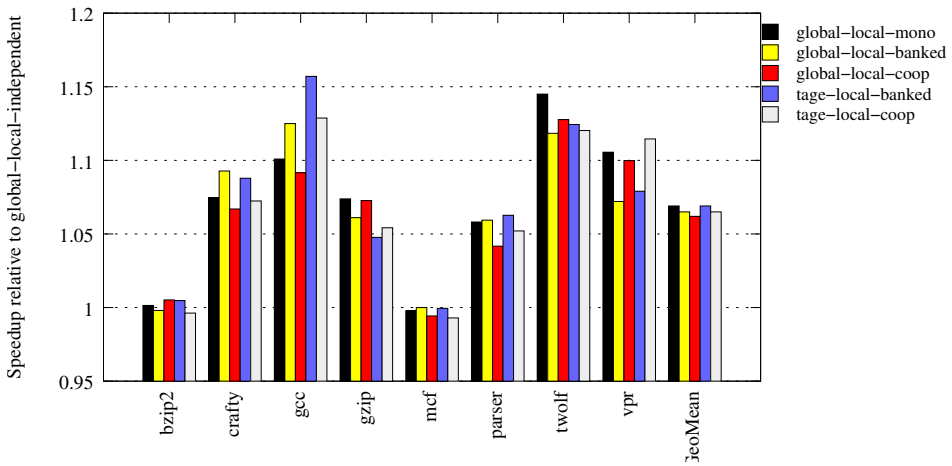
We now compare the relative impact of the four different distributed prediction schemes on performance. We present only integer benchmarks in these experiments as they show more MPKI variations across the four types of distributed predictors. We present results for a small TFlex processor configuration, the four-core TFlex and a larger TFlex processor configuration, the 16-core TFlex.

Figure 6.21 shows the speedups achieved by five different predictors over a baseline piecewise independent predictor using the local/global tournament exit predictor for four-core and 16-core configurations. The results are shown for a realistic TFlex configuration. Figure 6.22 shows the speedups achieved by the same five predictors over the baseline for four-core and 16-core TFlex configurations with perfect memory disambiguation. The five bars for each benchmark, from left to right, show the speedups achieved by a monolithic predictor using a local/global tournament exit predictor, a banked distributed predictor using a local/global tournament exit predictor, a co-operative distributed predictor using a local/global tournament exit predictor, a banked distributed predictor using a local/TAGE tournament predictor, and a co-operative distributed predictor using a local/TAGE tournament predictor.

The results for the four-core realistic configuration are disappointing. All the predictors achieve nearly the same speedups (about 2%) over the independent predictor with

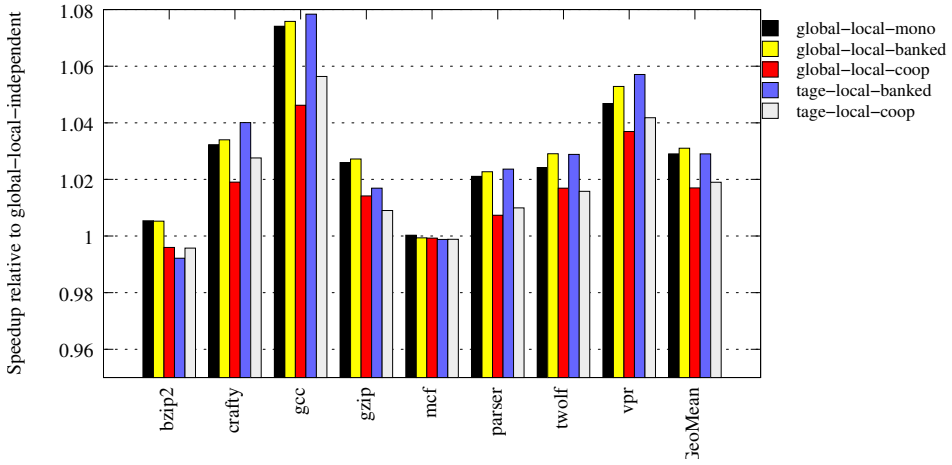


(a) four-core TFlex processor

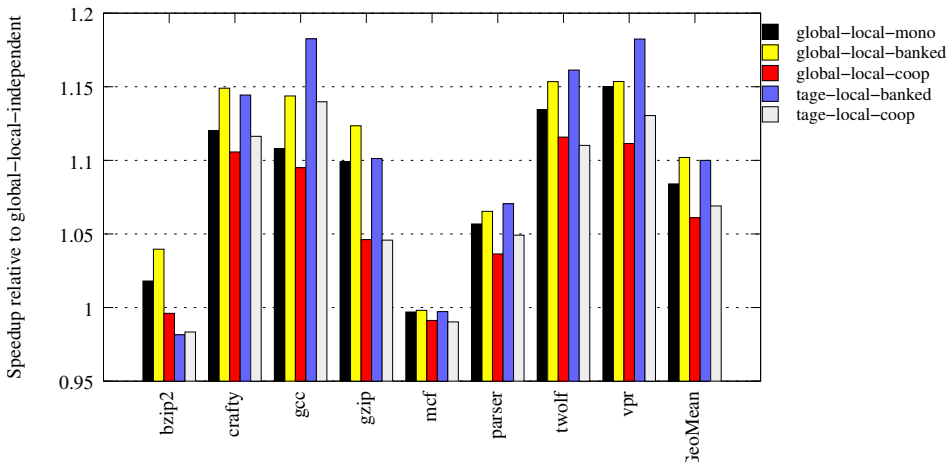


(b) 16-core TFlex processor

Figure 6.21: Speedups of SPEC integer benchmarks with five different predictors over a baseline piecewise independent predictor with local/global tournament exit predictor. Results are shown for four-core and 16-core realistic TFlex configurations. The five bars from left to right show the speedups of a monolithic predictor with local/global tournament exit predictor, banked distributed predictor with local/global tournament exit predictor, co-operative predictor with local/global tournament exit predictor, banked distributed predictor with local/TAGE tournament predictor, and co-operative predictor with local/TAGE tournament predictor.



(a) four-core TFlex processor



(b) 16-core TFlex processor

Figure 6.22: Speedups of SPEC integer benchmarks with five different predictors over a baseline piecewise independent predictor with local/global tournament exit predictor. Results are shown for four-core and 16-core TFlex configurations with perfect memory disambiguation. The five bars from left to right show the speedups of a monolithic predictor with local/global tournament exit predictor, banked distributed predictor with local/global tournament exit predictor, co-operative predictor with local/global tournament exit predictor, banked distributed predictor with local/TAGE tournament predictor, and co-operative predictor with local/TAGE tournament predictor.

the local/global co-operative predictor performing the worst. The best predictors are the monolithic predictor and the banked predictors. The local/TAGE co-operative predictor is slightly worse compared to the local/TAGE piecewise banked predictor. The MPKI improvement achieved by the co-operative predictor is not sufficiently high to offset the loss in performance incurred due to the extra prediction latency incurred in communicating with the neighboring cores to deliver the final prediction. The results for the 16-core configuration follow a similar trend except for the higher speedups obtained (about 7%). The low speedups over the independent design indicate that there are several bottlenecks to achieving higher performance in TFlex. When some of these bottlenecks are removed, the importance of accurate predictions will be higher.

We illustrate the above hypothesis by showing a TFlex configuration with perfect memory disambiguation. The relative MPKI values show a similar trend as before but the absolute values are slightly higher for a four-core configuration and much higher for the 16-core configuration. Other bottlenecks like OPN contention, cache misses, network latency, memory access latency are also challenges for TFlex. But this experiment illustrates that the benefit of a good predictor like the piecewise banked predictor compared to the piecewise independent predictor increases from 7% to 10% when removing the memory disambiguation bottleneck.

6.9 Communication latency and accuracy trade-offs

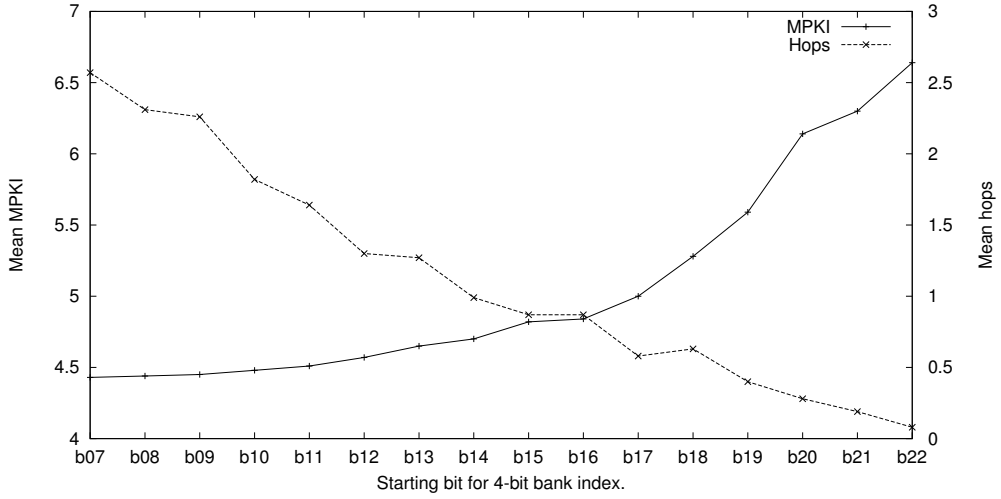
The banked distributed and co-operative distributed predictors described above require communication between owner cores for successive predictions and also between different cores to update the return address stack and return link stack. The co-operative predictors in our experiments require communication with immediate neighbors to collect individual predictions and then deliver the final prediction. In this section, we consider banked distributed predictors and compare the trade-offs of communication latency between predictions and prediction accuracy. We evaluate the impact of owner core choice . In our

earlier owner core experiments we showed how the owner core choice can impact static and dynamic block-to-bank distributions. We now evaluate the impact of owner core choice on the average hop count for communication between predictors in different cores and the impact on MPKI.

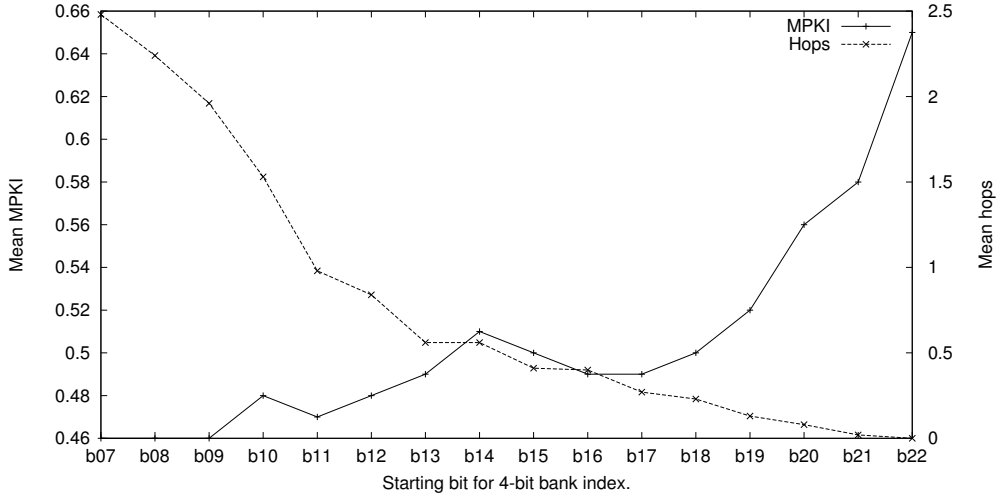
Since returns are not as commonly encountered as regular branches in the dynamic instruction stream in the benchmarks we evaluated, we expect the return address stack updates to be low overhead. Hence, we only consider owner core to owner core communication between predictions in the banked scheme. This communication is necessary to deliver the predicted next block address, updated global histories, and cached return address stack entries from the current owner core to the next owner core. The number of messages sent from one predictor to another can not only impact the predictor performance, but also have an impact on the control network congestion.

After presenting the static and dynamic block mappings, we chose to use the lowest order bits for generating the bank index for all our distributed predictor experiments. We now examine the effect of choosing different sets of bits for a 16-core TFlex configuration. Four consecutive bits are chosen for various starting bit positions from bit 7 to bit 22. The mean exit MPKI and the average hop count for various owner selections are illustrated in Figure 6.23 while the overall MPKI and the average hop count are shown in Figure 6.24.

When the owner core is chosen based on lower order bits, the ownership changes frequently and the block-to-bank mapping is more uniform. This helps the predictor use as much of the distributed state as possible and delivers lower MPKI. However, using the lower order for bank ownership results in frequent owner changes and after every prediction, the next owner will most likely be a different core. The chance of the same core being the owner again is low. Hence there is a relatively higher latency involved between successive block fetches in the processor to transfer the predictor information to the next owner core. This can lead to more network traffic and lower performance due to front-end bottlenecks. When higher order bits are chosen, there is less fetch-to-fetch delay but more skewed usage

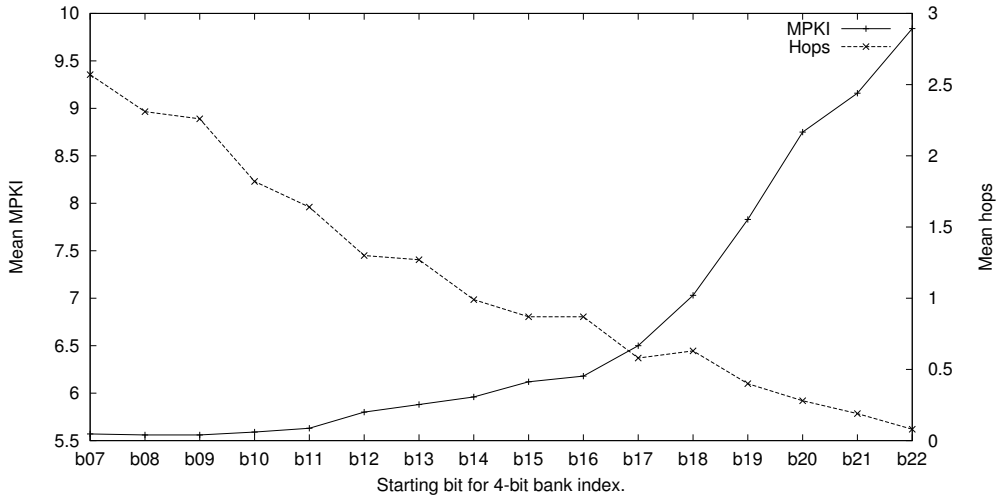


(a) SPEC2K integer benchmarks

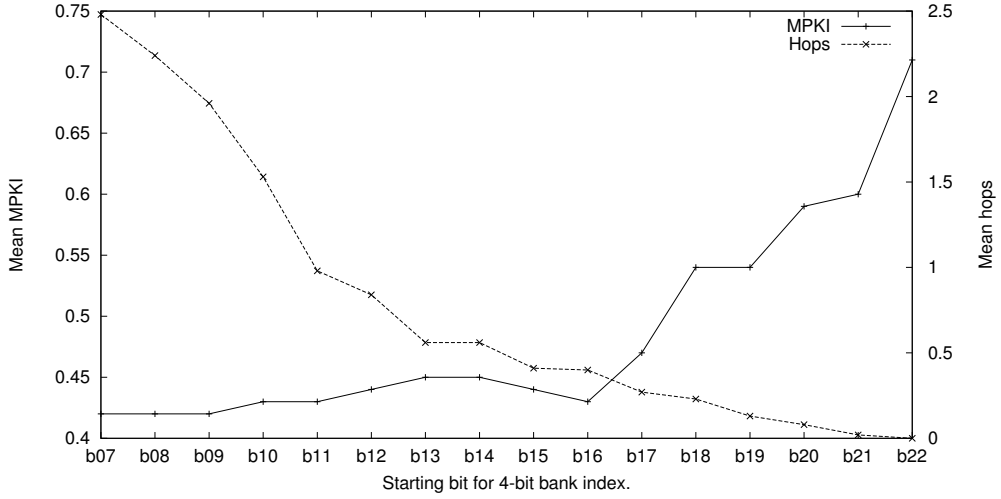


(b) SPEC2K FP benchmarks

Figure 6.23: Mean exit predictor MPKI and mean owner-to-owner communication hops for various bank ownership schemes determined by block address bits from bit 7 to bit 22. Results are shown for a 16-core TFlex configuration for integer and floating-point benchmarks. The predictor uses the banked distributed scheme with a local/global tournament exit predictor.



(a) SPEC2K integer benchmarks



(b) SPEC2K FP benchmarks

Figure 6.24: Mean overall predictor MPKI and mean owner-to-owner communication hops for various bank ownership schemes determined by block address bits from bit 7 to bit 22. Results are shown for a 16-core TFlex configuration for integer and floating-point benchmarks. The predictor uses the banked distributed scheme with a local/global tournament exit predictor.

of predictors in various cores. Hence there is a higher probability of mispredictions since the predictor state is not uniformly utilized. We present this trade-off by comparing exit and overall MPKI values with the hop count averages for the 16-core configuration.

Figure 6.23 shows that the average exit MPKI for the integer suite increases from 4.43 to 6.64 as the starting bit position for the owner is changed from the 7th bit to higher order bits up to the 22nd bit. On the contrary, the average owner-to-owner hop count goes down from 2.57 to 0.08. The MPKI increase is only marginal until the 11th bit. The average hop count goes down from 2.57 for the 7th bit to 1.64 for the 11th bit showing that the 11th bit as the starting bit (to determine the owner) is likely to be more beneficial than the currently used 7th bit position as it does not adversely affect the MPKI but reduces the average hop count by more than 36%. For the FP benchmarks, the MPKI increases from 0.46 to 0.65 while the average hop count goes down from 2.48 to 0. For the FP suite, there is very little increase in MPKI up to 18th bit position which shows an MPKI of 0.50 and average hop count of 0.23 (a reduction of about 64%).

We showed the exit MPKI separately to show how the exit predictor table usage (based on the owner core mapping) affects exit predictor MPKI and hop counts. Ultimately, the overall MPKI impacts the performance. Hence, we plot the overall MPKI and the average hop count in the Figure 6.24. The overall MPKI for the integer suite increases from 5.57 to 9.84 when moving from the 7th bit position for the owner to the 22nd bit position. For the FP benchmarks, the overall MPKI goes up from 0.42 to 0.71. When considering the overall MPKI we find that the best owner selection is the 11th bit position when considering the integer suite (with an MPKI of 5.63 and average hop count of 1.64) and the 17th bit position when considering the FP suite (with an MPKI of 0.47 and an average hop count of 1.27).

Since the integer benchmarks show significantly higher MPKI beyond the 12th bit position, we choose the 11th bit position as the best when considering both the benchmark suites. Hence, for a 16-core TFlex using the banked distributed prediction scheme, the

owner core bits can be chosen as bits[14:11] of the block address.

6.10 Summary

In this chapter we proposed distributed predictors for TFlex, a composable and flexible processor that has several lightweight cores. These predictors have the ability to adapt to the run-time ILP requirements of threads and provide high accuracies. We highlighted the difficulties in designing a small predictor and proposed various techniques to make use of predictors from several cores to construct a larger logical predictor. Considering the area, timing, and power constraints for the TFlex architecture, we came up with a 2.5 KB predictor for each core in TFlex. We presented a classification scheme for distributed prediction and highlighted the pros and cons of each design point. Predictors previously described in the earlier chapters were adapted to operate as distributed predictors for TFlex.

The independent distributed predictor had a simple design with low predictor accuracies. The banked distributed scheme and the co-operative scheme achieved the lowest MPKI and were comparable to the monolithic scheme. We showed predictor and performance results for each of the design points for various styles of predictors. We also experimented with various owner core choices and how they affect the average hop count and MPKI. When future chips include several small cores within a chip, the proposed techniques can enable execution of threads across several cores to achieve high single-thread performance.

Chapter 7

Related Work

In this chapter we discuss other efforts which are relevant to our prediction and analysis studies.

Multiple branch prediction and block prediction

Several previous studies have explored multiple branch and block prediction to enable high fetch bandwidth [8,11,61,80]. Most of these predictors were used to predict a small number of basic blocks (typically, two to four) at a time. These techniques are not very scalable beyond few basic blocks due to the probability of mispredictions being higher when deeper speculation is considered. Block prediction and exit prediction for block-based processors have been explored in [16, 21, 22, 47].

Exit prediction techniques for compiler-created blocks have been described previously. Hao et al [16] use exit predictors to predict the exits of extended basic blocks created by the compiler. They discuss predictor design alternatives and using variable-length exit encoding in the exit history. Multi-scalar [21, 70] processors use exit prediction followed by target prediction to predict next-task addresses. They discuss exit and path histories to make exit predictions and propose different alternatives for indexing the prediction table and maintaining the predicted exits in the prediction table. For target prediction, they

use stored targets in the header to predict direct branch targets, return address stack for returns, and propose a correlated task target buffer for indirect branches. Other prior efforts for block prediction have used the direct target prediction methodology instead of two-stage exit prediction followed by target prediction [22]. In our initial evaluations with the Trimaran [76] framework, direct target prediction methods were not as effective as exit prediction followed by target prediction for similar-sized predictors.

Prediction in block atomic and distributed architectures

To our knowledge, block-atomic call-return semantics and assumptions about branches to be made in distributed processors (number of exits in a block, type of branches, target addresses) have not been explored by previous studies. In Multiscalar [21, 70], several pieces of information about each exit (exit branch type, target address, and return address for calls and indirect calls) are stored in the task header to simplify targets prediction. In TRIPS/TFlex, we predict all these types and targets at run time instead of using header space. We did not find prior work on dynamically learning return addresses.

Branch type prediction has been discussed in the context of target prediction in high-frequency processors [83]. In several modern processors, the time to pre-decode and read the contents of a branch instruction is very small. Hence, many branch predictors choose to store the branch type information in the BTB. If the stored type is not a regular branch, another component in the target predictor makes the prediction. Though there are similarities between TRIPS and Wavescalar [73] in terms of data flow execution and distributed execution units, Wavescalar does not use speculation to fetch and map successive waves on to the wave cache.

Branch correlation and predictability

Correlation, branch behavior and limits of branch predictability have been explored in detail, especially for global predictors [6, 12]. They discuss why global correlation is im-

portant for predictability and the effect of long histories in capturing correlation. We are not aware of a similar analysis for predicated blocks. Prior to our work, Loh has used an interference-free perceptron to study the global correlation amongst branches in regular programs to motivate partitioned history predictors [36]. However, our use of perceptrons is to understand the global correlation loss that arises due to correlation-agnostic hyperblock construction heuristics.

Thomas et al. [75] used dataflow based value tracking across instructions to find the set of branches in the recent execution window correlated with a given branch (affectors). This information was then used to remove non-correlated branches from the global history for future predictions. Sazeides et al. [58] analyze the dynamic dataflow graph of programs and conclude that there is significant potential to exploit affector-affectee correlation to design efficient predictors. Tracking affectors and affectees can accurately show the branches that affect the direction of another branch.

Several researchers have examined the interaction of predication and branch predictability. Simon et al. discuss the notion of *misprediction migration*, where only one of a pair of correlated branches get predicated, thereby leaving the second branch without valuable correlation information in the dynamic history [64, 65]. Mantripragada et al. discuss a predictability-aware profiling-driven predication heuristic [40]. Stephenson et al. use several machine-learning techniques to make decisions on optimization heuristics in the compiler for intelligent predication [72]. They consider several parameters including predictability of individual branches. We are not aware of previous work that directly considers the effect of multiple correlated branches on predication.

Multiple predictors for distributed processors

There have been several architecture proposals which employ multiple-cores or multiple-execution units to execute a single thread.

Multiscalar [21, 70] processors use a ring of processing elements (PEs) each of

which executes a task (a portion of the program control flow graph). The tasks are fetched and committed in-order while execution of instructions within a task may be out-of-order. Multiscalar employs a hierarchical predictor with one next-task predictor in the processor and several intra-task branch predictors (one in each PE). The next-task predictor uses global histories and after every task prediction, it forwards its current history to the PE for that task. The branch predictor in that PE predicts the branches using the history forwarded by the task predictor. It also updates the history locally after every branch, to incorporate near-branch correlation information. Even though we do not employ such a hierarchical prediction scheme in TFlex, we showed that distributed predictors need to share the latest global histories to make effective predictions. Hence, we send the global histories from core to core to make predictions. Trace processors [55, 78] are similar to multiscalar processors but execute dynamically generated instruction traces on different processing units. Trace processors employ only next-trace predictors. As there is no internal control flow within a trace, there is no need for intra-trace branch prediction.

In Single-ISA heterogeneous multicore architectures [35], threads are assigned to cores based on their performance and power requirements. Typically more powerful cores have more complex and larger branch predictors which give higher accuracy compared to narrow-wide smaller cores which have static or simple dynamic branch predictors with lower accuracy.

The core fusion architecture [20, 20] uses a set of distributed predictors one present in each core. When multiple cores and "fused" together to form a larger logical core, the predictor structures from different cores are employed to make branch predictions. The predictor control logic and the fetch control logic are however global and not completely distributed as in the TFlex distributed prediction schemes. The global control initiates predictions across all cores, collects predictions, and makes decisions on the next set of instructions to be fetched. They also do not have a distributed RAS for target prediction. They choose to use the RAS in the first core to store return addresses. The argument for

this approach is that call-depth is a program property and increasing the RAS size while moving to multiple cores will not help programs when the call depth is low. The advantage is that the design of the RAS is simpler compared to the distributed RAS design in TFlex. However, the TFlex RAS has the ability to predict return addresses correctly even with long call chains found in object-oriented and recursive programs.

In [74], pairs of single-issue scalar cores are combined to form larger dual-issue out-of-order cores. They use simple next line predictors and bimodal predictors instead of more complex global history based schemes. Branch prediction for short threads in multi-threaded architectures is discussed in [7]. The authors discuss using correlation information from one thread to predict branches in the spawned threads better. The branch predictor of the spawned thread needs to start predicting with some useful history bits instead of using no history bits. Since complete history information from the parent thread and other threads may not be available, the authors propose the use of the initial program counter of the value of the spawned thread in the history. They find that this approach works almost as well as an idealized predictor using unknowable pre-history of the spawned speculative thread.

Chapter 8

Conclusions and Future Directions

Distributed block-based architectures pose several new challenges for control flow speculation. The challenges include block-level prediction made using one-of-eight exit prediction, prediction in the absence of branch type and target information at prediction time, handling block-atomic call-return semantics, and designing small but accurate distributed predictors for composable processors. The goals of this dissertation were to characterize these challenges and propose efficient techniques for accurate control flow prediction.

8.1 Summary of TRIPS prototype predictor

The TRIPS prototype predictor is described in Chapter 2. It has a two-stage prediction mechanism, exit prediction, followed by target address prediction. The prototype exit predictor is a scaled-down design of one of the best simple predictors from our early design space explorations. The TRIPS prototype required a predictor that was simple enough to design, implement, and verify in a short time frame, small enough to fit within the global tile, and very accurate in making block predictions. The prototype predictor occupies about 1.4 mm^2 area, uses slightly over 10 KB in storage, and employs single-ported structures in a simple three-cycle blocking design.

The predictor has a local/global tournament exit predictor and a multi-component target predictor. It has features like support for type prediction, block-atomic call-return semantics, speculative updates, and SMT mode prediction. The predictor achieves an average of 11.5% misprediction rate for SPEC integer benchmarks and 4.3% misprediction rate for SPEC floating-point benchmarks as measured on the prototype chip. The misprediction rates are reasonable but significantly higher than conventional branch predictors. Once the memory disambiguation and operand network contention bottlenecks were removed, we found that the speedup with perfect block prediction over the prototype prediction is over 20%. These results motivated our analysis of block prediction and predictability.

8.2 Summary of prediction analysis and better predictors

We started our analysis in Chapter 3 by looking at the misprediction (MPKI) breakdown of the 10 KB prototype predictor and the 32 KB scaled-up prototype predictor. The MPKI breakdown showed that the key sources of mispredictions were the exit predictor, the BTB, and the CTB. We performed exit prediction analysis from first principles to understand what kind of predictors can predict exits effectively.

Exit prediction

With four basic types of branch prediction components mapped to exit prediction (bimodal predictor, local predictor, global predictor, and path-based predictor) we evaluated various single and multiple-component predictors using these components. The scaling analysis showed that global and path-based predictors scale much better than local and bimodal predictors. We then studied aliasing effects which showed that most predictors up to 16 KB in size have severe aliasing in the prediction tables. Destructive aliasing can be potentially higher for exit predictors compared to branch predictors as the probability of the exit retrieved from an aliased entry being correct (one-in-eight) is lower than the probability of a branch retrieved from an aliased entry being correct (one-in-two). Multi-component pre-

dictors are typically better than equal-sized single component predictors. The local/global tournament predictor was the best among the evaluated two-component tournament predictors. Chooser inefficiencies result in a significant number of tournament predictor mispredictions. An imperfect chooser contributes to 26.8% of the exit mispredictions for the prototype predictor and 29.7% for the scaled-up 32 KB prototype predictor.

We evaluated many 16 KB multi-component predictors using different types of basic exit prediction components. The best predictor (with an ideal chooser) was the hybrid-4 predictor containing a 2 KB bimodal exit predictor component, a 4 KB local component, a 2 KB global component, and an 8 KB path-based component. This predictor with an ideal chooser achieved a very low MPKI of 1.81. Our next experiment was inspired from current state-of-the-art multi-component branch predictors which use different history lengths to effectively predict different branches in a program. We studied all possible one-component to nine-component interference-free global and path predictors (with ideal choosers) using nine different history lengths. Many of the best one-component to nine-component predictors had history lengths that showed an approximate geometric trend. This result motivated us to construct exit predictors inspired from OGEHL [60] and TAGE [62] branch predictors.

Using the insights from the analysis experiments, we evaluated better exit prediction techniques in Chapter 4. We studied stateless, table-based, and hybrid choosers for the hybrid-4 four-component predictor. We proposed an outcome-based chooser (inspired from fusion-based choosers [37]) that uses an efficient encoding of the relative outcomes of the four components along with the history to choose a component. This chooser outperformed the other choosers and achieved an exit MPKI reduction of 4% for the integer suite when compared to the 16 KB scaled-up prototype predictor.

To quantify the potential of long and multiple histories, we mapped three state-of-the-art branch predictors to exit prediction. The mapping of the OGEHL branch predictor to exit prediction did not outperform an equal-sized local/global tournament predictor because of inefficiencies in the exit predictor chooser component and table aliasing. Mapping of the

TAGE predictor is more straightforward since the final selection is based on tag match. We used address and path history hashed tags instead of the address-only tags in the original proposal. The TAGE exit predictor performed on par with the scaled-up tournament predictor while the local/TAGE predictor achieved a 4.9% lower integer MPKI for a 16 KB configuration and a 12.8% lower MPKI for a 64 KB configuration. These experiments also demonstrated the importance of using both longer global histories and per-block local histories. Finally, we mapped the piecewise perceptron predictor [24] to branch prediction by using eight perceptrons to make eight parallel binary predictions. The best asymmetric 8-perceptron predictor was only marginally better than the 16 KB scaled-up tournament predictor. However, the 64 KB asymmetric 8-perceptron showed an improvement of 15.5% for integer benchmarks compared to the 64 KB scaled-up tournament predictor.

Adapting efficient binary predictors like the OGEHL and the perceptron predictors to make 1-of-8 exit predictions was not very effective. To use branch predictors directly, we proposed the PPE (Post Prediction Encoding) predictor which used three predictors, one each to predict each bit of the exit. The final predicted exit ID is the concatenation of the three predicted bits. This predictor offers the flexibility to use different predictor types and sizes for each bit prediction. The asymmetric PPE predictor with the piecewise linear perceptron predictor as the component predictor was the best 16 KB exit predictor with an exit MPKI reduction of 8% for integer benchmarks and 6.8% for floating-point benchmarks when compared to the scaled-up prototype predictor. The best 64 KB predictor is the asymmetric 8-perceptron predictor with the asymmetric PPE predictor and the local/TAGE tournament predictor performing slightly inferior to the 8-perceptron predictor.

Exit predictability

Even the improved exit predictors were able to reduce only about 8% of the mispredictions compared to the simple local/global tournament predictor. All predictors containing global history-based components exploit global correlation across exits in hyperblocks. A compar-

ison of the prototype exit predictor (5 KB) for hyperblocks and a similar 5 KB local/global tournament branch predictor for basic blocks showed that the branch predictor had a 40% lower misprediction rate. It was inherently better at making predictions. To understand exit predictability, we analyzed the correlation between hyperblock exits.

Our tool used interference-free perceptrons to track the correlation between branches in the basic block code, mapped branches in the basic block code to exits and predicate-defines in the hyperblock code, and plotted where the dynamic strongly correlated branches lie in the hyperblock. We found that more than 50% of all the strongly correlated branches are present as predicate-defines, potentially indicating a severe loss in correlation. About 6% of the strongly correlated branches are present as exits in the same block and the remaining branches are present as exits in different blocks. For the correlated branches present as exits in the same block, the local predictor can effectively capture the per-block exit history. We also showed the usefulness of the local exit predictor component with a direct comparison to the local branch predictor component.

Target prediction

Other than the exit predictor, BTB and CTB mispredictions were significant contributors to the overall MPKI. For a few benchmarks, branch type and RAS mispredictions were seen. The branch type predictor and the CTB had severe table aliasing which was mostly removed when the target predictor was scaled from 5 KB to 16 KB. We had underestimated the branch offset length in the BTB and the return offset length in the CTB leading to several branch and return address mispredictions in the prototype predictor. Once these offset lengths were increased in the 16 KB target predictor, we found that most of the remaining mispredictions were due to indirect branches/calls present in three integer benchmarks.

We evaluated previously proposed techniques for indirect branch prediction that use global predictors, path-based predictors, and a state-of-the-art indirect branch predictor, the *ITTAGE* predictor [62]. Since two-level indirect predictor tables store the target addresses

in the second level table, the space requirements are high with increasing history lengths. Inspired by two-stage exit prediction followed by target prediction for blocks instead of direct target prediction, we designed an intermediate table of indirect-exits (between the histories and the indirect BTB) for indirect branch prediction to make more effective use of the predictor area. The indirect-exits were generated using a simple hash of the target address. We evaluated the indirect-exit based exit-inspired predictors in conjunction with several of the regular exit predictors like the path-based, global history-based, and TAGE exit predictors. We examined shifting in indirect-exit IDs also into the history to include the indirect target-based correlation information for future indirect branch predictions.

For small sizes the regular path history-based predictor performed well. For predictors of size 2 KB and beyond, the *ITTAGE* predictor and the exit-inspired predictors such as the *EIpath*, *EIshift*, and *EITAGE* predictors perform almost on par with each other. The addition of a small 1 KB exit-inspired path-based indirect branch predictor shows more than 20% reduction in MPKI for the entire integer suite. Simple predictors like the *EIpath* and *EIshift* can match the accuracy of sophisticated predictors like *ITTAGE* and *EITAGE* for over 2 KB sizes. For smaller sizes, the TAGE-based predictors are preferable.

Finally, we combined the best of the previously proposed enhancements for each component and compared the resulting predictor with the baseline 32 KB scaled-up local/global tournament predictor. The final improved predictor from our experiments had a 16 KB asymmetric PPE predictor (with piecewise-linear components), smaller branch type table (to compensate for the increase in storage in other components), three-bit branch types instead of two-bits to support indirect branches and calls, longer branch offsets in the BTB, longer return offsets in the CTB, smaller RAS (with extra bits in each entry to support the increased offset widths), and a 2 KB EITAGE indirect branch predictor. The MPKI went down from 6.49 to 4.06 for integer programs and went down from 1.3 to 1.14 for FP programs. Thus, the improved predictor showed an overall MPKI reduction of 37.4% for the integer suite and 13.8% for the FP suite. Even though the target predictor hides some of the

exit mispredictions (predicted target is correct even when predicted exit is wrong), the exit predictor contributes to most of the remaining mispredictions. About 66.5% of the overall mispredictions for integer benchmarks and 95.6% of the overall mispredictions for FP benchmarks are directly due to exit mispredictions.

8.3 Summary of distributed predictors

The TFlex distributed architecture imposes area, timing, and power constraints on the design of distributed predictors. The ability to operate as small but complete predictors in high TLP mode and as a logically larger predictor when multiple TFlex cores are combined to form a larger logical processor in high ILP mode was required. In Chapter 6, we proposed a classification for distributed processors that classified distributed predictors into four categories. We adapted several previously evaluated monolithic predictors for TRIPS to distributed prediction for TFlex. The predictors within each category differed in their exit prediction mechanism only. We considered predictors of size 2.5 KB (approximately) for each core in TFlex. Our experiments with different owner core choices showed that not using the lowermost address bits for the owner will help reduce the average owner-to-owner hop count while marginally increasing the MPKI.

The four categories described were the independent distributed predictor, the banked distributed predictor, the monolithic predictor, and the co-operative distributed predictor. The independent distributed predictor was scalable and easy to design but had the highest MPKI because of non-sharing of global histories across cores and use of the BTB instead of the RAS for return prediction. The banked distributed predictor resolved the shortcomings in the independent predictor by communicating the histories and cached RAS stack entries from the current owner core to the next owner core.

The monolithic predictor had a simple design with a centralized exit and target predictor. However, handling multiple requests at the same time and tolerating the extra latency for communicating with the centrally located predictor can make the predictor inefficient in

practice. The co-operative distributed predictor used predictor components from neighboring cores along with the components from the current owner core to make a prediction. The advantage of this predictor was that it could use sophisticated multi-component predictors and place different components of the same logical predictor in different cores. The best distributed predictor when considering MPKI was the local/TAGE co-operative distributed predictor with the banked distributed predictor following closely. Both these predictors achieved nearly the same MPKI as the monolithic predictor.

When considering the speedup compared to a baseline independent prediction scheme, all the other three predictors perform almost on par with each other. The best predictor is the local/TAGE banked distributed predictor which achieves about 7% speedup (10% with perfect memory disambiguation) compared to the baseline independent predictor. The MPKI improvement achieved by the co-operative distributed predictor is not sufficiently higher than the other predictors to hide the extra delay in the fetch pipeline incurred due to the latency of communicating with other cores to make the final prediction. For a TFlex configuration with perfect disambiguation and perfect operand network, perfect block prediction achieves a 52% speedup compared to realistic banked distributed block prediction scheme. When other TFlex bottlenecks are minimized, the predictor will be a key bottleneck to achieving higher performance.

8.4 Future directions and final thoughts

This dissertation showed how block predictors can be designed for distributed block-based architectures. Block predictors enable control flow speculation in distributed block-based architectures. However, improving block prediction alone does not guarantee effective speculation. Misprediction detection latency is crucial in reducing wasted cycles spent in executing wrong path instructions. Similarly, the misprediction recovery latency is also important as it indicates how soon the front-end can redirect the control flow and fill the pipeline with correct-path instructions.

With slightly more predictor storage and better sizing of tables, the prototype predictor could have achieved lower misprediction rates. If the storage budget is low, the local/global tournament exit predictor is still one of the best predictors. For larger designs, more sophisticated predictors like the local/TAGE predictor or the PPE predictor can be chosen. An indirect branch predictor component is necessary for high target prediction accuracy. Even a 1 KB indirect branch predictor can eliminate most of the indirect target mispredictions. Loop predictors and support for avoiding RAS overflows and underflows due to recursion may provide higher performance gains.

Several analysis results showed that ideal choosers achieve significantly lower MPKI compared to realistic choosers. Further exploration of choosers to find efficient but small choosers will help reduce exit prediction misprediction rates. Our improved exit and target prediction techniques can eliminate up to about a third of the mispredictions compared to the scaled-up prototype predictor. Optimized exit prediction techniques are necessary to remove most mispredictions. The local/TAGE predictor can be analyzed further with dynamic history and threshold fitting. New PPE predictor designs using different predictor types can be explored. Exit-inspired indirect branch predictors require further tuning to achieve low indirect branch/call MPKI at small sizes.

Block predictors can also be adapted to other architectures. For example, exit predictors can be used to predict first-taken branches in long instructions in VLIW processors. Similarly they can be used to predict fetched traces from trace cache-based superscalar processors. Exit prediction techniques can also be useful for predicting values. For example, if the values are within a small range, the PPE exit predictor can be employed for return value prediction, load value prediction, address prediction or way prediction. If there are many possible large values for the result of an instruction, a technique similar to the exit-inspired indirect prediction can be used where an intermediate table is used to select one of the predicted data values in the final value table.

Further work is needed to understand exit correlation better. We quantified how

much correlation is potentially lost due to predication and correlation-agnostic block formation. Future work to classify the predicate-defines into those that are required for correct prediction of the current branch and those that are not required for correct prediction (if another branch that is not predicated provides the correlation) will be beneficial in truly identifying the lost correlation. Despite this, our work indicates that correlation loss is partly responsible for exit mispredictions. To combat this problem, new exit prediction techniques that use information about predicate values should be evaluated. The hyperblock constructor can use profiled information about correlation between branches to intelligently place correlated branches as exits or predicate-defines. Another solution is to modify the compiler to provide exit IDs that indicate the predicated path through the block so that the exit history in the predictor can capture the predicate information also.

The piecewise banked and unified co-operative designs are the best among the four prediction mechanisms for distributed prediction in composable processors. Our analysis and experiments considered only scaled-down predictors that were used for monolithic prediction in TRIPS. New predictor designs tuned for distributed prediction may perform better in small sizes. Modeling control network contention will help understand the overhead of distributed predictor communication. We considered a TFlex execution model in which all the participating cores jointly execute every block. For other execution models [53] the predictor designs may have to be tuned differently. The distributed block prediction mechanisms are applicable to other block-based or instruction-based composable architectures where a logical larger processor is built by composing smaller cores together.

Any advances in branch prediction will be useful for monolithic and distributed block prediction. Power has become a first order constraint and hence future dies will have an increasing number of simple cores on a chip. However, these chips will be employed for both high ILP (for example, compilation or simulation) and high TLP (for example, web server) tasks. Distributed control flow predictors will be essential for future multi-core architectures that offer flexible power/performance operating points.

Bibliography

[1] J.R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Annual Symposium on Principles of Programming Languages*, pages 177–189, 1983.

[2] D. Burger, S.W. Keckler, K.S. McKinley, M. Dahlin, L.K. John, Calvin Lin, C.R. Moore, J. Burrill, R.G. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, Jul 2004.

[3] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[4] Po-Yung Chang, Eric Hao, and Yale N. Patt. Alternative implementations of hybrid branch predictors. In *international symposium on Microarchitecture*, pages 252–257, 1995.

[5] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)*, May 1997.

[6] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, 1996.

[7] Bumyong Choi, Leo Porter, and Dean M. Tullsen. Accurate branch prediction for short threads. pages 125–134, 2008.

[8] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *International Symposium on Computer Architecture*, pages 333–343, June 1995.

[9] Karel Driesen and Urs Holzle. Accurate indirect branch prediction. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, June 1998.

[10] Karel Driesen and Urs Holzle. The cascaded predictor: economical and adaptive branch target prediction. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[11] Simonjit Dutta and Manoj Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *International Symposium on Microarchitecture*, pages 258–263, December 1995.

[12] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *International Symposium on Computer Architecture*, pages 52–61, July 1998.

[13] Marus Evers, Po-Yung Chang, and Yale N. Patt. Using hybrid predictors to improve branch prediction accuracy in the presence of context switches. In *International Symposium on Computer Architecture*, pages 3–11, Jul 1996.

[14] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In *Proceedings of the 19th annual International Symposium on Computer architecture*, 1992.

[15] Mark Gebhart, Bertrand A. Maher, Katherine Coons, Jeff Diamond, Paul Gratz, Mario Marino, Nitya Ranganathan, Behnam Robatmili, Aaron Smith, James Burrill,

Stephen W. Keckler, Doug Burger, and Kathryn S. McKinley. An evaluation of the TRIPS computer system. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 1–12, March 2009.

[16] Eric Hao, Po-Yung Chang, Marius Evers, and Yale N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *International symposium on Microarchitecture*, pages 191–200, December 1996.

[17] Eric Hao, Po-Yung Chang, and Yale N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*, 1994.

[18] W.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Water, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G. Holm, and D.M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7:229–248, January 1993.

[19] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martnez. Accommodating workload diversity in chip multiprocessors via adaptive core fusion. In *Workshop on Complexity-effective Design*, Jun 2006.

[20] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martnez. Core fusion: accomodating software diversity in chip multiprocessors. In *International Symposium on Computer Architecture*, pages 186–197, May 2007.

[21] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *International Symposium on High Performance Computer Architecture*, pages 218–229, February 1997.

[22] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-based next trace pre-

diction. In *International Symposium on Microarchitecture*, pages 14–23, December 1997.

[23] Daniel Jimenez. Fast path-based neural branch prediction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[24] Daniel Jiménez. Piecewise linear branch prediction. In *International Symposium on Computer Architecture*, pages 382–393, Jun 2005.

[25] Daniel Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *International Symposium on High Performance Computer Architecture*, pages 197–206, January 2001.

[26] Daniel Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. *ACM Transactions on Computer Systems*, 20(4):369–397, 2002.

[27] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. Recovery requirements of branch prediction storage structures in the presence of mispredicted-path execution. *International Journal of Parallel Programming*, 25, Oct 1997.

[28] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. pages 34–42, 1991.

[29] Richard E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[30] Changkyu Kim, Simha Sethumadhavan, M. S. Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W. Keckler. Composable lightweight processors. In *International Symposium on Microarchitecture*, pages 381–394, Dec 2007.

[31] Changkyu Kim, Simha Sethumadhavan, Nitya Ranganathan, Haiming Liu, Robert McDonald, Doug Burger, and Stephen W. Keckler. Elastic threads on composable

processors. Technical Report TR-06-09, Department of Computer Sciences, The University of Texas at Austin, Mar 2006.

[32] Chankyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, Oct 2002.

[33] Hyesoon Kim, Jose A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert Cohn. Vpc prediction: Reducing the cost of indirect branches via hardware-based dynamic virtualization. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 424–435, 2007.

[34] Hyesoon Kim, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2d-profiling: detecting input-dependent branches with a single input data set. In *International Symposium on Code Generation and Optimization*, pages 159–172, 2006.

[35] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *International Symposium on Microarchitecture*, pages 81–92, 2003.

[36] Gabriel H. Loh. A simple divide-and-conquer approach for neural-class branch prediction. In *Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[37] Gabriel H. Loh and Dana S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[38] Bertrand A. Maher, Aaron Smith, Doug Burger, and Kathryn S. McKinley. Head and

tail duplication for convergent hyperblock formation. In *International Symposium on Microarchitecture*, pages 65–76, Dec 2006.

[39] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Microarchitecture*, pages 45–54, 1992.

[40] Srinivas Mantripragada and Alexandru Nicolau. Using profiling to reduce branch misprediction costs on a dynamically scheduled processor. In *International Conference on Supercomputing*, pages 206–214, 2000.

[41] Scott McFarling. Combining branch predictors. Technical Report TN-36, DEC WRL, Jun 1993.

[42] Stephen W. Melvin, Michael C. Shebanow, and Yale Patt. Hardware support for large atomic units in dynamically scheduled machines. In *21st Annual International Workshop on Microprogramming and Microarchitecture*, November 1988.

[43] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, April 1965.

[44] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *International Symposium on Microarchitecture*, pages 40–51, Dec 2001.

[45] Ravi Nair. Dynamic path-based branch correlation. In *International Symposium on Microarchitecture*, pages 15–23, 1995.

[46] Mario D. Nemirovsky, Forrest Brewer, and Roger C. Wood. Disc: dynamic instruction stream computer. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 163–171, November 1991.

[47] Sanjay J. Patel and Steven S. Lumetta. replay: A hardware framework for dynamic optimization. *IEEE Transactions of Computers*, Jun 2001.

[48] Sanjay J. Patel, Tony Tung, Satarupa Bose, and Matthew M. Crum. Increasing the size of atomic instruction blocks using control flow assertions. In *In Proceedings of the 33th Annual International Symposium on Microarchitecture*, pages 303–313, 2000.

[49] Nitya Ranganathan, Doug Burger, and Stephen W. Keckler. Analysis of the TRIPS prototype block predictor. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2009.

[50] Nitya Ranganathan, Ramadass Nagarajan, Daniel Jiménez, Doug Burger, Stephen W. Keckler, and Calvin Lin. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, Sep 2002.

[51] Glenn Reinman, Todd Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the 26th annual international symposium on Computer architecture*, pages 234–245, May 1999.

[52] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd annual international symposium on Microarchitecture*, pages 16–27, November 1999.

[53] Behnam Robatmili, Katherine E. Coons, Doug Burger, and Kathryn S. McKinley. Strategies for mapping dataflow blocks to distributed hardware. In *Proceedings of the 41st International Symposium on Microarchitecture*, pages 23–34, November 2008.

[54] Franziska Roesner, Doug Burger, and Stephen W. Keckler. Counting dependence predictors. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 215–226, 2008.

[55] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and James E. Smith. Trace processors. In *International Symposium on Microarchitecture*, December 1997.

[56] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *International Symposium on Computer Architecture*, pages 422–433, Jun 2003.

[57] Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, Madhu Saravana Sibi Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethmadhavan, Sadia Sharif, Premkishore Shivakumar, Stephen W. Keckler, and Doug Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *International Symposium on Microarchitecture*, pages 480–491, Dec 2006.

[58] Yiannakis Sazeides, Andreas Moustakas, Kypros Constantinides, and Marios Kleanthous. The significance of affectors and affectees correleations for branch prediction. *Lecture Notes in Computer Science*, 4917/2008:243–257, 2008.

[59] Simha Sethumadhavan, Franziska Roesner, Joel S. Emer, Doug Burger, and Stephen W. Keckler. Late-binding: enabling unordered load-store queues. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 347–357, 2007.

[60] Andre Seznec. Analysis of the O-GEmetric History Length branch predictor. In *International Symposium on Computer Architecture*, pages 394–405, Jun 2005.

[61] Andre Seznec, Stephan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Architectural Support for Programming Languages and Operating Systems*, pages 116–127, 1996.

[62] Andre Seznec and Pierre Michaud. A case for (partially) tagged geometric history length predictor. *Journal of Instruction-level Parallelism*, Feb 2006.

[63] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block istribution analysis to find periodic behavior and simulation points in applications. In *Internationl Symposium on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[64] Beth Simon, Brad Calder, and Jeanne Ferrante. Incorporating predicate information into branch predictors. In *Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Techonology*, December 2001.

[65] Beth Simon, Brad Calder, and Jeanne Ferrante. Incorporating predicate information into branch predictors. In *International Symposium on High Performance Computer Architecture*, pages 53–64, Feb 2003.

[66] K. Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, December 1998.

[67] K. Skadron, M. Martonosi, and D. Clark. Speculative updates of local and global branch history: A quantitative analysis. *Journal of Instruction-level Parallelism*, 2, January 2000.

[68] Aaron Smith, Jim Burrill, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, and Kathryn S. McKinley. Compiling for EDGE architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, March 2006.

[69] James E. Smith. A study of branch prediction strategies. In *International Symposium on Computer Architecture*, pages 135–148, May 1981.

[70] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 414–425, June 1995.

[71] Jared Stark, Marius Evers, and Yale N. Patt. Variable length path branch prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 170–179, Oct 1998.

[72] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *ACM SIGPLAN Conference on Programming Language design and Implementation*, pages 77–90, 2003.

[73] Steve Swanson, Ken Michaelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *International Symposium on Microarchitecture*, Dec 2003.

[74] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: repurposing scalar cores for out-of-order instruction issue. In *Proceedings of the 45th Annual Conference on Design automation*, pages 772–775, 2008.

[75] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *International Symposium on Computer Architecture*, pages 314–323, Jun 2003.

[76] Trimaran: An infrastructure for research in instruction-level parallelism. http://www.trimaran.org.

[77] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[78] Sriram Vajapeyam and Tulika Mitra. Improving superscalar instruction dispatch and

issue by exploiting dynamic code sequences. In *International Symposium on Computer Architecture*, pages 1–12, May 1997.

[79] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirosky. Performance estimation of multistreamed, superscalar processors. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, pages 195–204, January 1994.

[80] T.-Y. Yeh, D. Marr, and Y. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *International Conference on Supercomputing*, pages 67–76, Jul 1993.

[81] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *International Symposium on Microarchitecture*, pages 51–61, 1991.

[82] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch prediction. In *International Symposium on Computer Architecture*, pages 124–134, 1992.

[83] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *International symposium on Microarchitecture*, pages 129–139, 1992.

Vita

Nitya Ranganathan was born in Paramakkudi, India to V.R.S. Ranganathan and R. Anjana. She graduated from the TVS Lakshmi Matriculation Higher Secondary School at Madurai in 1997 and enrolled in the College of Engineering, Guindy, Anna University at Chennai. She earned the Bachelor of Engineering in Computer Science and Engineering in May, 2001. In the fall of 2001, she joined the Ph.D. program in Computer Sciences at the University of Texas at Austin. While pursuing her Ph.D. degree, she obtained the degree of Master of Science in Computer Sciences in May, 2005.

Permanent Address: Old no. B2, 44, New no. 97/10
Second Main Road
Gandhi Nagar, Adyar
Chennai, India 600020.

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.